Sage Reference Manual: 3D Graphics

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

EXAMPLES:

```
sage: x, y = var('x y')
sage: W = plot3d(sin(pi*((x)^2+(y)^2))/2,(x,-1,1),(y,-1,1), frame=False, color='purple', opacity=0.8)
sage: S = sphere((0,0,0),size=0.3, color='red', aspect_ratio=[1,1,1])
sage: show(W + S, figsize=8)
```

CHAPTER

TWO

PARAMETRIC PLOTS

sage.plot.plot3d.parametric_plot3d.**parametric_plot3d**(*f*, *urange*, *vrange=None*, *plot_points='automatic'*, *boundary_style=None*, ***kwds*)

Return a parametric three-dimensional space curve or surface.

There are four ways to call this function:

- `parametric_plot3d([f_x, f_y, f_z], (u_min, u_max))`: f_x, f_y, f_z are three functions and $u_{\min}$ and $u_{\max}$ are real numbers
- `parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max))`: f_x, f_y, f_z can be viewed as functions of u
- `parametric_plot3d([f_x, f_y, f_z], (u_min, u_max), (v_min, v_max))`: f_x, f_y, f_z are each functions of two variables
- `parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max), (v, v_min, v_max))`: f_x, f_y, f_z can be viewed as functions of u and v

INPUT:

- `f` - a 3-tuple of functions or expressions, or vector of size 3
- `urange` - a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- `vrange` - (optional - only used for surfaces) a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)
- `plot_points` - (default: "automatic", which is 75 for curves and [40,40] for surfaces) initial number of sample points in each parameter; an integer for a curve, and a pair of integers for a surface.
- `boundary_style` - (default: None, no boundary) a dict that describes how to draw the boundaries of regions by giving options that are passed to the line3d command.
- `mesh` - bool (default: False) whether to display mesh grid lines
- `dots` - bool (default: False) whether to display dots at mesh grid points

---

Note:

1. By default for a curve any points where f_x, f_y, or f_z do not evaluate to a real number are skipped.
2. Currently for a surface f_x, f_y, and f_z have to be defined everywhere. This will change.
3. mesh and dots are not supported when using the Tachyon ray tracer renderer.

EXAMPLES: We demonstrate each of the four ways to call this function.

1.A space curve defined by three functions of 1 variable:

```
sage: parametric_plot3d( (sin, cos, lambda u: u/10), (0, 20))
```

Note above the lambda function, which creates a callable Python function that sends u to $u/10$.

2.Next we draw the same plot as above, but using symbolic functions:

```
sage: u = var('u')
sage: parametric_plot3d( (sin(u), cos(u), u/10), (u, 0, 20))
```

3.We draw a parametric surface using 3 Python functions (defined using lambda):

```
sage: f = (lambda u,v: cos(u), lambda u,v: sin(u)+cos(v), lambda u,v: sin(v))
sage: parametric_plot3d(f, (0, 2*pi), (-pi, pi))
```

4.The surface, but with a mesh:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi), mesh=
```

5.The same surface, but where the defining functions are symbolic:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi))
```

We increase the number of plot points, and make the surface green and transparent:

```
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi), color
```

We call the space curve function but with polynomials instead of symbolic variables.

```
sage: R.<t> = RDF[]
sage: parametric_plot3d( (t, t^2, t^3), (t, 0, 3) )
```

Next we plot the same curve, but because we use (0, 3) instead of (t, 0, 3), each polynomial is viewed as a callable function of one variable:

```
sage: parametric_plot3d( (t, t^2, t^3), (0, 3) )
```

We do a plot but mix a symbolic input, and an integer:

```
sage: t = var('t')
sage: parametric_plot3d( (1, sin(t), cos(t)), (t, 0, 3) )
```

We specify a boundary style to show us the values of the function at its extrema:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, pi), (v, 0, pi), \
...                     boundary_style={"color": "black", "thickness": 2})
```

We can plot vectors:

```
sage: x,y=var('x,y')
sage: parametric_plot3d(vector([x-y,x*y,x*cos(y)]), (x,0,2), (y,0,2))
sage: t=var('t')
sage: p=vector([1,2,3])
sage: q=vector([2,-1,2])
sage: parametric_plot3d(p*t+q, (t, 0, 2))
```

Any options you would normally use to specify the appearance of a curve are valid as entries in the boundary_style dict.

MANY MORE EXAMPLES:

We plot two interlinked tori:

```
sage: u, v = var('u,v')
sage: f1 = (4+(3+cos(v))*sin(u), 4+(3+cos(v))*cos(u), 4+sin(v))
sage: f2 = (8+(3+cos(v))*cos(u), 3+sin(v), 4+(3+cos(v))*sin(u))
sage: p1 = parametric_plot3d(f1, (u,0,2*pi), (v,0,2*pi), texture="red")
sage: p2 = parametric_plot3d(f2, (u,0,2*pi), (v,0,2*pi), texture="blue")
sage: p1 + p2
```

A cylindrical Star of David:

```
sage: u,v = var('u v')
sage: f_x = cos(u)*cos(v)*(abs(cos(3*v/4))^500 + abs(sin(3*v/4))^500)^(-1/260)*(abs(cos(4*u/4))^
sage: f_y = cos(u)*sin(v)*(abs(cos(3*v/4))^500 + abs(sin(3*v/4))^500)^(-1/260)*(abs(cos(4*u/4))^
sage: f_z = sin(u)*(abs(cos(4*u/4))^200 + abs(sin(4*u/4))^200)^(-1/200)
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, 0, 2*pi))
```

Double heart:

```
sage: u, v = var('u,v')
sage: f_x = ( abs(v) - abs(u) - abs(tanh((1/sqrt(2))*u)/(1/sqrt(2))) + abs(tanh((1/sqrt(2))*v)/(
sage: f_y = ( abs(v) - abs(u) - abs(tanh((1/sqrt(2))*u)/(1/sqrt(2))) - abs(tanh((1/sqrt(2))*v)/(
sage: f_z = sin(u)*(abs(cos(4*u/4))^1 + abs(sin(4*u/4))^1)^(-1/1)
sage: parametric_plot3d([f_x, f_y, f_z], (u, 0, pi), (v, -pi, pi))
```

Heart:

```
sage: u, v = var('u,v')
sage: f_x = cos(u)*(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
sage: f_y = sin(u) *(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
sage: f_z = v
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, -1, 1), frame=False, color="red")
```

Green bowtie:

```
sage: u, v = var('u,v')
sage: f_x = sin(u) / (sqrt(2) + sin(v))
sage: f_y = sin(u) / (sqrt(2) + cos(v))
sage: f_z = cos(u) / (1 + sqrt(2))
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, -pi, pi), frame=False, color="green")
```

Boy's surface http://en.wikipedia.org/wiki/Boy's_surface

```
sage: u, v = var('u,v')
sage: fx = 2/3* (cos(u)* cos(2*v) + sqrt(2)* sin(u)* cos(v))* cos(u) / (sqrt(2) - sin(2*u)* sin(
sage: fy = 2/3* (cos(u)* sin(2*v) - sqrt(2)* sin(u)* sin(v))* cos(u) / (sqrt(2) - sin(2*u)* sin(
sage: fz = sqrt(2)* cos(u)* cos(u) / (sqrt(2) - sin(2*u)* sin(3*v))
sage: parametric_plot3d([fx, fy, fz], (u, -2*pi, 2*pi), (v, 0, pi), plot_points = [90,90], frame
```

Maeder's_Owl (pretty but can't find an internet reference):

```
sage: u, v = var('u,v')
sage: fx = v *cos(u) - 0.5* v^2 * cos(2* u)
sage: fy = -v *sin(u) - 0.5* v^2 * sin(2* u)
sage: fz = 4 *v^1.5 * cos(3 *u / 2) / 3
sage: parametric_plot3d([fx, fy, fz], (u, -2*pi, 2*pi), (v, 0, 1),plot_points = [90,90], frame=F
```

Bracelet:

```
sage: u, v = var('u,v')
sage: fx = (2 + 0.2*sin(2*pi*u))*sin(pi*v)
sage: fy = 0.2*cos(2*pi*u) *3*cos(2*pi*v)
sage: fz = (2 + 0.2*sin(2*pi*u))*cos(pi*v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, pi/2), (v, 0, 3*pi/4), frame=False, color="gray")
```

Green goblet

```
sage: u, v = var('u,v')
sage: fx = cos(u)*cos(2*v)
sage: fy = sin(u)*cos(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, pi), frame=False, color="green")
```

Funny folded surface - with square projection:

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(2*v)
sage: fy = sin(u)*cos(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

Surface of revolution of figure 8:

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(2*v)
sage: fy = sin(u)*sin(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

Yellow Whitney's umbrella http://en.wikipedia.org/wiki/Whitney_umbrella:

```
sage: u, v = var('u,v')
sage: fx = u*v
sage: fy = u
sage: fz = v^2
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -1, 1), frame=False, color="yellow")
```

Cross cap http://en.wikipedia.org/wiki/Cross-cap:

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
sage: fz = -tanh((2/3)*(u-pi))*sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Twisted torus:

```
sage: u, v = var('u,v')
sage: fx = (3+sin(v)+cos(u))*cos(2*v)
sage: fy = (3+sin(v)+cos(u))*sin(2*v)
sage: fz = sin(u)+2*cos(v)
```

```
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Four intersecting discs:

```
sage: u, v = var('u,v')
sage: fx = v *cos(u) -0.5*v^2*cos(2*u)
sage: fy = -v*sin(u) -0.5*v^2*sin(2*u)
sage: fz = 4* v^1.5 *cos(3* u / 2) / 3
sage: parametric_plot3d([fx, fy, fz], (u, 0, 4*pi), (v, 0,2*pi), frame=False, color="red", opaci
```

Steiner surface/Roman's surface (see http://en.wikipedia.org/wiki/Roman_surface and http://en.wikipedia.org/wiki/Steiner_surface):

```
sage: u, v = var('u,v')
sage: fx = (sin(2 * u) * cos(v) * cos(v))
sage: fy = (sin(u) * sin(2 * v))
sage: fz = (cos(u) * sin(2 * v))
sage: parametric_plot3d([fx, fy, fz], (u, -pi/2, pi/2), (v, -pi/2,pi/2), frame=False, color="red
```

Klein bottle? (see http://en.wikipedia.org/wiki/Klein_bottle):

```
sage: u, v = var('u,v')
sage: fx = (3*(1+sin(v)) + 2*(1-cos(v)/2)*cos(u))*cos(v)
sage: fy = (4+2*(1-cos(v)/2)*cos(u))*sin(v)
sage: fz = -2*(1-cos(v)/2) * sin(u)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

A Figure 8 embedding of the Klein bottle (see http://en.wikipedia.org/wiki/Klein_bottle):

```
sage: u, v = var('u,v')
sage: fx = (2 + cos(v/2)* sin(u) - sin(v/2)* sin(2 *u))* cos(v)
sage: fy = (2 + cos(v/2)* sin(u) - sin(v/2)* sin(2 *u))* sin(v)
sage: fz = sin(v/2)* sin(u) + cos(v/2) *sin(2* u)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Enneper's surface (see http://en.wikipedia.org/wiki/Enneper_surface):

```
sage: u, v = var('u,v')
sage: fx = u -u^3/3  + u*v^2
sage: fy = v -v^3/3  + v*u^2
sage: fz = u^2 - v^2
sage: parametric_plot3d([fx, fy, fz], (u, -2, 2), (v, -2, 2), frame=False, color="red")
```

Henneberg's surface (see http://xahlee.org/surface/gallery_m.html)

```
sage: u, v = var('u,v')
sage: fx = 2*sinh(u)*cos(v) -(2/3)*sinh(3*u)*cos(3*v)
sage: fy = 2*sinh(u)*sin(v) +(2/3)*sinh(3*u)*sin(3*v)
sage: fz = 2*cosh(2*u)*cos(2*v)
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -pi/2, pi/2), frame=False, color="red")
```

Dini's spiral

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(v)
sage: fy = sin(u)*sin(v)
sage: fz = (cos(v)+log(tan(v/2))) + 0.2*u
sage: parametric_plot3d([fx, fy, fz], (u, 0, 12.4), (v, 0.1, 2),frame=False, color="red")
```

Catalan's surface (see http://xahlee.org/surface/catalan/catalan.html):

```
sage: u, v = var('u,v')
sage: fx = u-sin(u)*cosh(v)
sage: fy = 1-cos(u)*cosh(v)
sage: fz = 4*sin(1/2*u)*sinh(v/2)
sage: parametric_plot3d([fx, fy, fz], (u, -pi, 3*pi), (v, -2, 2), frame=False, color="red")
```

A Conchoid:

```
sage: u, v = var('u,v')
sage: k = 1.2; k_2 = 1.2; a = 1.5
sage: f = (k^u*(1+cos(v))*cos(u), k^u*(1+cos(v))*sin(u), k^u*sin(v)-a*k_2^u)
sage: parametric_plot3d(f, (u,0,6*pi), (v,0,2*pi), plot_points=[40,40], texture=(0,0.5,0))
```

A Mobius strip:

```
sage: u,v = var("u,v")
sage: parametric_plot3d([cos(u)*(1+v*cos(u/2)), sin(u)*(1+v*cos(u/2)), 0.2*v*sin(u/2)], (u,0, 4*
```

A Twisted Ribbon

```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 3*sin(u)*sin(v), cos(v)], (u,0, 2*pi), (v, 0, pi),plot
```

An Ellipsoid:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 2*sin(u)*sin(v), cos(u)], (u,0, 2*pi), (v, 0, 2*pi),pl
```

A Cone:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u], (u, -1, 1), (v, 0, 2*pi+0.5), plot_points=[50,5
```

A Paraboloid:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u^2], (u, 0, 1), (v, 0, 2*pi+0.4), plot_points=[50,
```

A Hyperboloid:

```
sage: u, v = var('u,v')
sage: plot3d(u^2-v^2, (u, -1, 1), (v, -1, 1), plot_points=[50,50])
```

A weird looking surface - like a Mobius band but also an O:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([sin(u)*cos(u)*log(u^2)*sin(v), (u^2)^(1/6)*(cos(u)^2)^(1/4)*cos(v), sin
```

A heart, but not a cardioid (for my wife):

```
sage: u, v = var('u,v')
sage: p1 = parametric_plot3d([sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/20)*(cos(u)^2)^(1/4)-1
sage: p2 = parametric_plot3d([-sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/20)*(cos(u)^2)^(1/4)-
sage: show(p1+p2, frame=False)
```

A Hyperhelicoidal:

```
sage: u = var("u")
sage: v = var("v")
sage: fx = (sinh(v)*cos(3*u))/(1+cosh(u)*cosh(v))
sage: fy = (sinh(v)*sin(3*u))/(1+cosh(u)*cosh(v))
```

```
sage: fz = (cosh(v)*sinh(u))/(1+cosh(u)*cosh(v))
sage: parametric_plot3d([fx, fy, fz], (u, -pi, pi), (v, -pi, pi), plot_points = [50,50], frame=F
```

A Helicoid (lines through a helix, http://en.wikipedia.org/wiki/Helix):

```
sage: u, v = var('u,v')
sage: fx = sinh(v)*sin(u)
sage: fy = -sinh(v)*cos(u)
sage: fz = 3*u
sage: parametric_plot3d([fx, fy, fz], (u, -pi, pi), (v, -pi, pi), plot_points = [50,50], frame=F
```

Kuen's surface (http://virtualmathmuseum.org/Surface/kuen/kuen.html):

```
sage: fx = (2*(cos(u) + u*sin(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: fy = (2*(sin(u) - u*cos(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: fz = log(tan(1/2 *v)) + (2*cos(v))/(1+ u^2*sin(v)^2)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0.01, pi-0.01), plot_points = [50,50], f
```

A 5-pointed star:

```
sage: fx = cos(u)*cos(v)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)*(abs(cos(5*v/4))^1
sage: fy = cos(u)*sin(v)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)*(abs(cos(5*v/4))^1
sage: fz = sin(u)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)
sage: parametric_plot3d([fx, fy, fz], (u, -pi/2, pi/2), (v, 0, 2*pi), plot_points = [50,50], fra
```

A cool self-intersecting surface (Eppener surface?):

```
sage: fx = u - u^3/3 + u*v^2
sage: fy = v - v^3/3 + v*u^2
sage: fz = u^2 - v^2
sage: parametric_plot3d([fx, fy, fz], (u, -25, 25), (v, -25, 25), plot_points = [50,50], frame=F
```

The breather surface (http://en.wikipedia.org/wiki/Breather_surface):

```
sage: fx = (2*sqrt(0.84)*cosh(0.4*u)*(-(sqrt(0.84)*cos(v)*cos(sqrt(0.84)*v)) - sin(v)*sin(sqrt(0
sage: fy = (2*sqrt(0.84)*cosh(0.4*u)*(-(sqrt(0.84)*sin(v)*cos(sqrt(0.84)*v)) + cos(v)*sin(sqrt(0
sage: fz = -u + (2*0.84*cosh(0.4*u)*sinh(0.4*u))/(0.4*((sqrt(0.84)*cosh(0.4*u))^2 + (0.4*sin(sqr
sage: parametric_plot3d([fx, fy, fz], (u, -13.2, 13.2), (v, -37.4, 37.4), plot_points = [90,90],
```

TESTS:

```
sage: u, v = var('u,v')
sage: plot3d(u^2-v^2, (u, -1, 1), (u, -1, 1))
Traceback (most recent call last):
...
ValueError: range variables should be distinct, but there are duplicates
```

From Trac #2858:

```
sage: parametric_plot3d((u,-u,v), (u,-10,10),(v,-10,10))
sage: f(u)=u; g(v)=v^2; parametric_plot3d((g,f,f), (-10,10),(-10,10))
```

From Trac #5368:

```
sage: x, y = var('x,y')
sage: plot3d(x*y^2 - sin(x), (x,-1,1), (y,-1,1))
```

CHAPTER

THREE

PARAMETRIC SURFACE

Parametric Surface

Graphics 3D object for triangulating surfaces, and a base class for many other objects that can be represented by a 2D parametrization.

It takes great care to turn degenerate quadrilaterals into triangles and to propagate identified points to all attached polygons. This is not so much to save space as it is to assist the raytracers/other rendering systems to better understand the surface (and especially calculate correct surface normals).

AUTHORS:

- Robert Bradshaw (2007-08-26): initial version

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface, MobiusStrip
sage: def f(x,y): return x+y, sin(x)*sin(y), x*y
sage: P = ParametricSurface(f, (srange(0,10,0.1), srange(-5,5.0,0.1)))
sage: show(P)
sage: S = MobiusStrip(1,.2)
sage: S.is_enclosed()
False
sage: S.show()
```

Note: One may override `eval()` or `eval_c()` in a subclass rather than passing in a function for greater speed. One also would want to override get_grid.

TODO: actually remove unused points, fix the below code:

```
S = ParametricSurface(f=(lambda (x,y):(x,y,0)), domain=(range(10),range(10)))
```

class sage.plot.plot3d.parametric_surface.**MobiusStrip**(*r*, *width*, *twists=1*, ***kwds*)

Bases: sage.plot.plot3d.parametric_surface.ParametricSurface

Base class for the MobiusStrip graphics type. This sets the the basic parameters of the object.

INPUT:

- `r` - A number which can be coerced to a float, serving roughly as the radius of the object.
- `width` - A number which can be coerced to a float, which gives the width of the object.
- `twists` - (default: 1) An integer, giving the number of twists in the object (where one twist is the 'traditional' Mobius strip).

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: M = MobiusStrip(3,3)
sage: M.show()
```

eval(*u*, *v*)

Returns tuple for x, y, z coordinates for the given u and v for this MobiusStrip instance.

EXAMPLE:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: N = MobiusStrip(7,3,2) # two twists
sage: N.eval(-1,0)
(4.0, 0.0, -0.0)
```

get_grid(*ds*)

Returns appropriate u and v ranges for this MobiusStrip instance. This is intended for internal use in creating an actual plot.

INPUT:

- ds - A number, typically coming from a RenderParams object, which helps determine the increment for the v range for the MobiusStrip object.

EXAMPLE:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: N = MobiusStrip(7,3,2) # two twists
sage: N.get_grid(N.default_render_params().ds)
([-1, 1], [0.0, 0.125663706144, 0.251327412287, 0.376991118431, ... 0])
```

class sage.plot.plot3d.parametric_surface.**ParametricSurface**

Bases: sage.plot.plot3d.index_face_set.IndexFaceSet

Base class that initializes the ParametricSurface graphics type. This sets options, the function to be plotted, and the plotting array as attributes.

INPUT:

- f - (default: None) The defining function. Either a tuple of three functions, or a single function which returns a tuple, taking two python floats as input. To subclass, pass None for f and override eval_c or eval instead.
- domain - (default: None) A tuple of two lists, defining the grid of u, v values. If None, this will be calculate automatically.

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface
sage: def f(x,y): return cos(x)*sin(y), sin(x)*sin(y), cos(y)+log(tan(y/2))+0.2*x
sage: S = ParametricSurface(f, (srange(0,12.4,0.1), srange(0.1,2,0.1)))
sage: show(S)

sage: len(S.face_list())
2214
```

The Hessenberg surface:

```
sage: def f(u,v):
...       a = 1
...       from math import cos, sin, sinh, cosh
...       x = cos(a)*(cos(u)*sinh(v)-cos(3*u)*sinh(3*v)/3) + sin(a)*(
```

```
...             sin(u)*cosh(v)-sin(3*u)*cosh(3*v)/3)
...         y = cos(a)*(sin(u)*sinh(v)+sin(3*u)*sinh(3*v)/3) + sin(a)*(
...             -cos(u)*cosh(v)-cos(3*u)*cosh(3*v)/3)
...         z = cos(a)*cos(2*u)*cosh(2*v)+sin(a)*sin(2*u)*sinh(2*v)
...         return (x,y,z)
sage: v = srange(float(0),float((3/2)*pi),float(0.1))
sage: S = ParametricSurface(f, (srange(float(0),float(pi),float(0.1)),
...                     srange(float(-1),float(1),float(0.1))), color="blue")
sage: show(S)
```

bounding_box()

Returns the lower and upper corners of a 3D bounding box for self. This is used for rendering and self should fit entirely within this box.

Specifically, the first point returned should have x, y, and z coordinates should be the respective infimum over all points in self, and the second point is the supremum.

EXAMPLES:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: M = MobiusStrip(7,3,2)
sage: M.bounding_box()
((-10.0, -7.53907349250478..., -2.9940801852848145), (10.0, 7.53907349250478..., 2.994080185
```

default_render_params()

Returns an instance of RenderParams suitable for plotting this object.

TEST:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: type(MobiusStrip(3,3).default_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

dual()

Returns an `IndexFaceSet` which is the dual of the `ParametricSurface` object as a triangulated surface.

EXAMPLES:

As one might expect, this gives an icosahedron:

```
sage: D = dodecahedron()
sage: D.dual()
```

But any enclosed surface should work:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: T =  Torus(1, .2)
sage: T.dual()
sage: T.is_enclosed()
True
```

Surfaces which are not enclosed, though, should raise an exception:

```
sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: M = MobiusStrip(3,1)
sage: M.is_enclosed()
False
sage: M.dual()
Traceback (most recent call last):
```

```
...
NotImplementedError: This is only implemented for enclosed surfaces
```

eval(*u*, *v*)

TEST:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface
sage: def f(x,y): return x+y,x-y,x*y
sage: P = ParametricSurface(f,(srange(0,1,0.1),srange(0,1,0.1)))
sage: P.eval(0,0)
Traceback (most recent call last):
...
NotImplementedError
```

get_grid(*ds*)

TEST:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface
sage: def f(x,y): return x+y,x-y,x*y
sage: P = ParametricSurface(f)
sage: P.get_grid(.1)
Traceback (most recent call last):
...
NotImplementedError: You must override the get_grid method.
```

is_enclosed()

Returns a boolean telling whether or not it is necessary to render the back sides of the polygons (assuming, of course, that they have the correct orientation).

This is calculated in by verifying the opposite edges of the rendered domain either line up or are pinched together.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Sphere
sage: Sphere(1).is_enclosed()
True

sage: from sage.plot.plot3d.parametric_surface import MobiusStrip
sage: MobiusStrip(1,0.2).is_enclosed()
False
```

jmol_repr(*render_params*)

Returns representation of the object suitable for plotting using Jmol.

TESTS:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
sage: s = P.jmol_repr(P.testing_render_params())
sage: s[:10]
['pmesh obj_1 "obj_1.pmesh"\ncolor pmesh  [102,102,255]']
```

json_repr(*render_params*)

Returns representation of the object in JSON format as a list with one element, which is a string of a dictionary listing vertices and faces.

TESTS:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
```

```
sage: s = P.json_repr(P.default_render_params())
sage: s[0][:100]
'{vertices:[{x:-2,y:-2,z:0},{x:-2,y:-1.89744,z:0.399737},{x:-2,y:-1.79487,z:0.778435},{x:-2,
```

obj_repr(*render_params*)

Returns complete representation of object with name, texture, and lists of vertices, faces, and back-faces.

TESTS:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
sage: s = P.obj_repr(P.default_render_params())
sage: s[:2]+s[2][:3]+s[3][:3]
['g obj_1', 'usemtl texture...', 'v -2 -2 0', 'v -2 -1.89744 0.399737', 'v -2 -1.79487 0.778
```

tachyon_repr(*render_params*)

Returns representation of the object suitable for plotting using Tachyon ray tracer.

TESTS:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
sage: s = P.tachyon_repr(P.default_render_params())
sage: s[:2]
['TRI V0 -2 -2 0 V1 -2 -1.89744 0.399737 V2 -1.89744 -1.89744 0', 'texture...']
```

triangulate(*render_params=None*)

Calls self.eval_grid() for all (u, v) in urange $\times$ vrange to construct this surface.

The most complicated part of this code is identifying shared vertices and shrinking trivial edges. This is not done so much to save memory, rather it is needed so normals of the triangles can be calculated correctly.

TESTS:

```
sage: from sage.plot.plot3d.parametric_surface import ParametricSurface, MobiusStrip
sage: def f(x,y): return x+y, sin(x)*sin(y), x*y                          # indirect doctests
sage: P = ParametricSurface(f, (srange(0,10,0.1), srange(-5,5.0,0.1)))   # indirect doctests
sage: P.show()                                                            # indirect doctests
sage: S = MobiusStrip(1,.2)                                               # indirect doctests
sage: S.show()                                                            # indirect doctests
```

x3d_geometry()

Returns XML-like representation of the coordinates of all points in a triangulation of the object along with an indexing of those points.

TESTS:

```
sage: _ = var('x,y')
sage: P = plot3d(x^2-y^2, (x, -2, 2), (y, -2, 2))
sage: s = P.x3d_str()
sage: s[:100]
"<Shape>\n<IndexedFaceSet coordIndex='0,1,..."
```

CHAPTER

FOUR

SURFACES OF REVOLUTION

AUTHORS:

- Oscar Gerardo Lazo Arjona (2010): initial version.

sage.plot.plot3d.revolution_plot3d.**revolution_plot3d**(*curve*, *trange*, *phirange=None*, *parallel_axis='z'*, *axis=(0, 0)*, *print_vector=False*, *show_curve=False*, ***kwds*)

Return a plot of a revolved curve.

There are three ways to call this function:

- `revolution_plot3d(f,trange)` where f is a function located in the xz plane.
- `revolution_plot3d((f_x,f_z),trange)` where (f_x, f_z) is a parametric curve on the xz plane.
- `revolution_plot3d((f_x,f_y,f_z),trange)` where (f_x, f_y, f_z) can be any parametric curve.

INPUT:

- `curve` - A curve to be revolved, specified as a function, a 2-tuple or a 3-tuple.
- `trange` - A 3-tuple $(t, t_{\min}, t_{\max})$ where t is the independent variable of the curve.
- `phirange` - A 2-tuple of the form $(\phi_{\min}, \phi_{\max})$, (default $(0, \pi)$) that specifies the angle in which the curve is to be revolved.
- `parallel_axis` - A string (Either 'x', 'y', or 'z') that specifies the coordinate axis parallel to the revolution axis.
- `axis` - A 2-tuple that specifies the position of the revolution axis. If parallel is:
  - 'z' - then axis is the point in which the revolution axis intersects the xy plane.
  - 'x' - then axis is the point in which the revolution axis intersects the yz plane.
  - 'y' - then axis is the point in which the revolution axis intersects the xz plane.
- `print_vector` - If True, the parametrization of the surface of revolution will be printed.
- `show_curve` - If True, the curve will be displayed.

EXAMPLES:

Let's revolve a simple function around different axes:

```
sage: u = var('u')
sage: f=u^2
sage: revolution_plot3d(f,(u,0,2),show_curve=True,opacity=0.7).show(aspect_ratio=(1,1,1))
```

If we move slightly the axis, we get a goblet-like surface:

```
sage: revolution_plot3d(f,(u,0,2),axis=(1,0.2),show_curve=True,opacity=0.5).show(aspect_ratio=(1
```

A common problem in calculus books, find the volume within the following revolution solid:

```
sage: line=u
sage: parabola=u^2
sage: sur1=revolution_plot3d(line,(u,0,1),opacity=0.5,rgbcolor=(1,0.5,0),show_curve=True,paralle
sage: sur2=revolution_plot3d(parabola,(u,0,1),opacity=0.5,rgbcolor=(0,1,0),show_curve=True,paral
sage: (sur1+sur2).show()
```

Now let's revolve a parametrically defined circle. We can play with the topology of the surface by changing the axis, an axis in $(0,0)$ (as the previous one) will produce a sphere-like surface:

```
sage: u = var('u')
sage: circle=(cos(u),sin(u))
sage: revolution_plot3d(circle,(u,0,2*pi),axis=(0,0),show_curve=True,opacity=0.5).show(aspect_ra
```

An axis on $(0,y)$ will produce a cylinder-like surface:

```
sage: revolution_plot3d(circle,(u,0,2*pi),axis=(0,2),show_curve=True,opacity=0.5).show(aspect_ra
```

And any other axis will produce a torus-like surface:

```
sage: revolution_plot3d(circle,(u,0,2*pi),axis=(2,0),show_curve=True,opacity=0.5).show(aspect_ra
```

Now, we can get another goblet-like surface by revolving a curve in 3d:

```
sage: u = var('u')
sage: curve=(u,cos(4*u),u^2)
sage: revolution_plot3d(curve,(u,0,2),show_curve=True,parallel_axis='z',axis=(1,.2),opacity=0.5)
```

A curvy curve with only a quarter turn:

```
sage: u = var('u')
sage: curve=(sin(3*u),.8*cos(4*u),cos(u))
sage: revolution_plot3d(curve,(u,0,pi),(0,pi/2),show_curve=True,parallel_axis='z',opacity=0.5).s
```

CHAPTER

FIVE

PLOTTING 3D FIELDS

sage.plot.plot3d.plot_field3d.**plot_vector_field3d**(*functions*, *xrange*, *yrange*, *zrange*, *plot_points=5*, *colors='jet'*, *center_arrows=False*, ***kwds*)

Plot a 3d vector field

INPUT:

- `functions` - a list of three functions, representing the x-, y-, and z-coordinates of a vector
- `xrange`, `yrange`, and `zrange` - three tuples of the form (var, start, stop), giving the variables and ranges for each axis
- `plot_points` (default 5) - either a number or list of three numbers, specifying how many points to plot for each axis
- `colors` (default 'jet') - a color, list of colors (which are interpolated between), or matplotlib colormap name, giving the coloring of the arrows. If a list of colors or a colormap is given, coloring is done as a function of length of the vector
- `center_arrows` (default False) - If True, draw the arrows centered on the points; otherwise, draw the arrows with the tail at the point
- any other keywords are passed on to the plot command for each arrow

EXAMPLES:

```
sage: x,y,z=var('x y z')
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi))
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),colors=['red
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),colors='red'
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),plot_points=
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),plot_points=
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),center_arrow
```

TESTS:

This tests that trac ticket #2100 is fixed in a way compatible with this command:

```
sage: plot_vector_field3d((x*cos(z),-y*cos(z),sin(z)), (x,0,pi), (y,0,pi), (z,0,pi),center_arrow
```

CHAPTER

SIX

IMPLICIT PLOTS

sage.plot.plot3d.implicit_plot3d.**implicit_plot3d**(*f*, *xrange*, *yrange*, *zrange*, ***kwds*)

Plots an isosurface of a function.

INPUT:

- `f` - function
- `xrange` - a 2-tuple (x_min, x_max) or a 3-tuple (x, x_min, x_max)
- `yrange` - a 2-tuple (y_min, y_may) or a 3-tuple (y, y_min, y_may)
- `zrange` - a 2-tuple (z_min, z_maz) or a 3-tuple (z, z_min, z_maz)
- `plot_points` - (default: "automatic", which is 50) the number of function evaluations in each direction. (The number of cubes in the marching cubes algorithm will be one less than this). Can be a triple of integers, to specify a different resolution in each of x,y,z.
- `contour` - (default: 0) plot the isosurface f(x,y,z)==contour. Can be a list, in which case multiple contours are plotted.
- `region` - (default: None) If region is given, it must be a Python callable. Only segments of the surface where region(x,y,z) returns a number >0 will be included in the plot. (Note that returning a Python boolean is acceptable, since True == 1 and False == 0).

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
```

A simple sphere:

```
sage: implicit_plot3d(x^2+y^2+z^2==4, (x, -3, 3), (y, -3,3), (z, -3,3))
```

A nested set of spheres with a hole cut out:

```
sage: implicit_plot3d((x^2 + y^2 + z^2), (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=60, con
...                   region=lambda x,y,z: x<=0.2 or y>=0.2 or z<=0.2).show(viewer='tachyon')
```

A very pretty example, attributed to Douglas Summers-Stay (archived page):

```
sage: T = RDF(golden_ratio)
sage: p = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z - T*x) + cos(z
sage: r = 4.77
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40).show(viewer='tachyo
```

As I write this (but probably not as you read it), it's almost Valentine's day, so let's try a heart (from http://mathworld.wolfram.com/HeartSurface.html)

```
sage: p = (x^2+9/4*y^2+z^2-1)^3-x^2*z^3-9/(80)*y^2*z^3
sage: r = 1.5
sage: implicit_plot3d(p, (x, -r,r), (y, -r,r), (z, -r,r), plot_points=80, color='red', smooth=Fa
```

The same examples also work with the default Jmol viewer; for example:

```
sage: T = RDF(golden_ratio)
sage: p = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z - T*x) + cos(z
sage: r = 4.77
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40).show()
```

Here we use smooth=True with a Tachyon graph:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, smooth=Tru
```

We explicitly specify a gradient function (in conjunction with smooth=True) and invert the normals:

```
sage: gx = lambda x, y, z: -(2*x + y^2 + z^2)
sage: gy = lambda x, y, z: -(x^2 + 2*y + z^2)
sage: gz = lambda x, y, z: -(x^2 + y^2 + 2*z)
sage: implicit_plot3d(x^2+y^2+z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, \
...       plot_points=40, smooth=True, gradient=(gx, gy, gz)).show(viewer='tachyon')
```

A graph of two metaballs interacting with each other:

```
sage: def metaball(x0, y0, z0): return 1 / ((x-x0)^2 + (y-y0)^2 + (z-z0)^2)
sage: implicit_plot3d(metaball(-0.6, 0, 0) + metaball(0.6, 0, 0), (x, -2, 2), (y, -2, 2), (z, -2
```

MANY MORE EXAMPLES:

A kind of saddle:

```
sage: implicit_plot3d(x^3 + y^2 - z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=60, contc
```

A smooth surface with six radial openings:

```
sage: implicit_plot3d(-(cos(x) + cos(y) + cos(z)), (x, -4, 4), (y, -4, 4), (z, -4, 4))
```

A cube composed of eight conjoined blobs:

```
sage: implicit_plot3d(x^2 + y ^2 + z^2 +cos(4*x)+cos(4*y)+cos(4*z)-0.2, (x, -2, 2), (y, -2, 2),
```

A variation of the blob cube featuring heterogeneously sized blobs:

```
sage: implicit_plot3d(x^2 + y ^2 + z^2 +sin(4*x) + sin(4*y) + sin(4*z) -1, (x, -2, 2), (y, -2, 2
```

A klein bottle:

```
sage: implicit_plot3d((x^2+y^2+z^2+2*y-1)*((x^2+y^2+z^2-2*y-1)^2-8*z^2)+16*x*z*(x^2+y^2+z^2-2*y-
```

A lemniscate:

```
sage: implicit_plot3d(4*x^2*(x^2+y^2+z^2+z)+y^2*(y^2+z^2-1), (x, -0.5, 0.5), (y, -1, 1), (z, -1,
```

Drope:

```
sage: implicit_plot3d(z - 4*x*exp(-x^2-y^2), (x, -2, 2), (y, -2, 2), (z, -1.7, 1.7))
```

A cube with a circular aperture on each face:

```
sage: implicit_plot3d(((1/2.3)^2 *(x^2 + y^2 + z^2))^-6 + ( (1/2)^8 * (x^8 + y^8 + z^8) )^6 -1,
```

A simple hyperbolic surface:

```
sage: implicit_plot3d(x*x + y - z*z, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A hyperboloid:

```
sage: implicit_plot3d(x^2 + y^2 - z^2 -0.3, (x, -2, 2), (y, -2, 2), (z, -1.8, 1.8))
```

Duplin cycloid:

```
sage: implicit_plot3d((2^2 - 0^2 - (2 + 2.1)^2) * (2^2 - 0^2 - (2 - 2.1)^2)*(x^4+y^4+z^4)+ 2*((2
```

Sinus:

```
sage: implicit_plot3d(sin(pi*((x)^2+(y)^2))/2 +z, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A torus:

```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^2 + z*z - 1, (x, -4, 4), (y, -4, 4), (z, -1, 1))
```

An octahedron:

```
sage: implicit_plot3d(abs(x)+abs(y)+abs(z) - 1, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A cube:

```
sage: implicit_plot3d(x^100 + y^100 + z^100 -1, (x, -2, 2), (y, -2, 2), (z, -2, 2))
```

Toupie:

```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^3 + z*z - 1, (x, -4, 4), (y, -4, 4), (z, -6, 6))
```

A cube with rounded edges:

```
sage: implicit_plot3d(x^4 + y^4 + z^4 - (x^2 + y^2 + z^2), (x, -2, 2), (y, -2, 2), (z, -2, 2))
```

Chmutov:

```
sage: implicit_plot3d(x^4 + y^4 + z^4 - (x^2 + y^2 + z^2-0.3), (x, -1.5, 1.5), (y, -1.5, 1.5), (
```

Further Chutmov:

```
sage: implicit_plot3d(2*(x^2*(3-4*x^2)^2+y^2*(3-4*y^2)^2+z^2*(3-4*z^2)^2) -3, (x, -1.3, 1.3), (y
```

Clebsch:

```
sage: implicit_plot3d(81*(x^3+y^3+z^3)-189*(x^2*y+x^2*z+y^2*x+y^2*z+z^2*x+z^2*y) +54*x*y*z+126*(
```

Looks like a water droplet:

```
sage: implicit_plot3d(x^2 +y^2 -(1-z)*z^2, (x, -1.5, 1.5), (y, -1.5, 1.5), (z, -1, 1))
```

Sphere in a cage:

```
sage: implicit_plot3d((x^8 + z^30 + y^8 - (x^4 + z^50 + y^4 -0.3))*(x^2 + y^2 + z^2 -0.5), (x, -
```

Ortho circle:

```
sage: implicit_plot3d(((x^2 + y^2 - 1)^2 + z^2)* ((y^2 + z^2 - 1)^2 + x^2)* ((z^2 + x^2 - 1)^2 +
```

Cube sphere:

```
sage: implicit_plot3d(12 - ((1/2.3)^2 *(x^2 + y^2 + z^2))^-6 - ( (1/2)^8 * (x^8 + y^8 + z^8) )^6
```

Two cylinders intersect to make a cross:

```
sage: implicit_plot3d((x^2 + y^2 - 1) * ( x^2 + z^2 - 1) - 1, (x, -3, 3), (y, -3, 3), (z, -3, 3)
```

Three cylinders intersect in a similar fashion:

```
sage: implicit_plot3d((x^2 + y^2 - 1) * ( x^2 + z^2 - 1)* ( y^2 + z^2 - 1) - 1, (x, -3, 3), (y,
```

A sphere-ish object with twelve holes, four on each XYZ plane:

```
sage: implicit_plot3d(3*(cos(x) + cos(y) + cos(z)) + 4* cos(x) * cos(y) * cos(z), (x, -3, 3), (y
```

A gyroid:

```
sage: implicit_plot3d(cos(x) * sin(y) + cos(y) * sin(z) + cos(z) * sin(x), (x, -4, 4), (y, -4, 4
```

Tetrahedra:

```
sage: implicit_plot3d((x^2 + y^2 + z^2)^2 + 8*x*y*z - 10*(x^2 + y^2 + z^2) + 25, (x, -4, 4), (y,
```

TESTS:

Test a separate resolution in the X direction; this should look like a regular sphere:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=(10, 40,
```

Test using different plot ranges in the different directions; each of these should generate half of a sphere. Note that we need to use the `aspect_ratio` keyword to make it look right with the unequal plot ranges:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, 0, 2), (y, -2, 2), (z, -2, 2), contour=4, aspect_rati

sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, 0, 2), (z, -2, 2), contour=4, aspect_rati

sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, 0, 2), contour=4, aspect_rati
```

Extra keyword arguments will be passed to show():

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, viewer='ta
```

An implicit plot that doesn't include any surface in the view volume produces an empty plot:

```
sage: implicit_plot3d(x^2 + y^2 + z^2 - 5000, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=6)
```

Make sure that implicit_plot3d doesn't error if the function cannot be symbolically differentiated:

```
sage: implicit_plot3d(max_symbolic(x, y^2) - z, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=
```

CHAPTER

SEVEN

LIST PLOTS

sage.plot.plot3d.list_plot3d.**list_plot3d**(*v*, *interpolation_type='default'*, *texture='automatic'*, *point_list=None*, ***kwds*)

A 3-dimensional plot of a surface defined by the list v of points in 3-dimensional space.

INPUT:

- `v` - something that defines a set of points in 3 space, for example:
  - a matrix
  - a list of 3-tuples
  - a list of lists (all of the same length) - this is treated the same as a matrix.
- `texture` - (default: "automatic", a solid light blue)

OPTIONAL KEYWORDS:

- `interpolation_type` - 'linear', 'nn' (nearest neighbor), 'spline'

  'linear' will perform linear interpolation

  The option 'nn' will interpolate by averaging the value of the nearest neighbors, this produces an interpolating function that is smoother than a linear interpolation, it has one derivative everywhere except at the sample points.

  The option 'spline' interpolates using a bivariate B-spline.

  When v is a matrix the default is to use linear interpolation, when v is a list of points the default is nearest neighbor.

- `degree` - an integer between 1 and 5, controls the degree of spline used for spline interpolation. For data that is highly oscillatory use higher values
- `point_list` - If point_list=True is passed, then if the array is a list of lists of length three, it will be treated as an array of points rather than a 3xn array.
- `num_points` - Number of points to sample interpolating function in each direction, when `interpolation_type` is not `default`. By default for an $n \times n$ array this is n.
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

We plot a matrix that illustrates summation modulo n.

```
sage: n = 5; list_plot3d(matrix(RDF,n,[(i+j)%n for i in [1..n] for j in [1..n]]))
```

We plot a matrix of values of sin.

```
sage: pi = float(pi)
sage: m = matrix(RDF, 6, [sin(i^2 + j^2) for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, texture='yellow', frame_aspect_ratio=[1,1,1/3])
```

Though it doesn't change the shape of the graph, increasing num_points can increase the clarity of the graph.

```
sage: list_plot3d(m, texture='yellow', frame_aspect_ratio=[1,1,1/3],num_points=40)
```

We can change the interpolation type.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='nn',frame_aspect_ratio=[1,1,1/3])
```

We can make this look better by increasing the number of samples.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='nn',frame_aspect_ratio=[1,1,1/3],num_
```

Let's try a spline.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='spline',frame_aspect_ratio=[1,1,1/3])
```

That spline doesn't capture the oscillation very well; let's try a higher degree spline.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='spline', degree=5, frame_aspect_ratio
```

We plot a list of lists:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]]))
```

We plot a list of points. As a first example we can extract the (x,y,z) coordinates from the above example and make a list plot out of it. By default we do linear interpolation.

```
sage: l=[]
sage: for i in range(6):
...       for j in range(6):
...          l.append((float(i*pi/5),float(j*pi/5),m[i,j]))
sage: list_plot3d(l,texture='yellow')
```

Note that the points do not have to be regularly sampled. For example:

```
sage: l=[]
sage: for i in range(-5,5):
...     for j in range(-5,5):
...       l.append((normalvariate(0,1),normalvariate(0,1),normalvariate(0,1)))
sage: list_plot3d(l,interpolation_type='nn',texture='yellow',num_points=100)
```

TESTS:

We plot 0, 1, and 2 points:

```
sage: list_plot3d([])

sage: list_plot3d([(2,3,4)])

sage: list_plot3d([(0,0,1), (2,3,4)])
```

However, if two points are given with the same x,y coordinates but different z coordinates, an exception will be raised:

```
sage: pts =[(-4/5, -2/5, -2/5), (-4/5, -2/5, 2/5), (-4/5, 2/5, -2/5), (-4/5, 2/5, 2/5), (-2/5, -
sage: show(list_plot3d(pts, interpolation_type='nn'))
Traceback (most recent call last):
```

```
...
ValueError: Two points with same x,y coordinates and different z coordinates were given. Interpo
```

Additionally we need at least 3 points to do the interpolation:

```
sage: mat = matrix(RDF, 1, 2, [3.2, 1.550])
sage: show(list_plot3d(mat,interpolation_type='nn'))
Traceback (most recent call last):
...
ValueError: We need at least 3 points to perform the interpolation
```

sage.plot.plot3d.list_plot3d.**list_plot3d_array_of_arrays**(*v*, *interpolation_type*, *texture*, ***kwds*)

A 3-dimensional plot of a surface defined by a list of lists `v` defining points in 3-dimensional space. This is done by making the list of lists into a matrix and passing back to `list_plot3d()`. See `list_plot3d()` for full details.

INPUT:

- `v` - a list of lists, all the same length
- `interpolation_type` - (default: 'linear')
- `texture` - (default: "automatic", a solid light blue)

OPTIONAL KEYWORDS:

- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

The resulting matrix does not have to be square:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1]])) # indirect doctest
```

The normal route is for the list of lists to be turned into a matrix and use `list_plot3d_matrix()`:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]]))
```

With certain extra keywords (see `list_plot3d_matrix()`), this function will end up using `list_plot3d_tuples()`:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]],interpolation_ty
```

sage.plot.plot3d.list_plot3d.**list_plot3d_matrix**(*m*, *texture*, ***kwds*)

A 3-dimensional plot of a surface defined by a matrix `M` defining points in 3-dimensional space. See `list_plot3d()` for full details.

INPUT:

- `M` - a matrix
- `texture` - (default: "automatic", a solid light blue)

OPTIONAL KEYWORDS:

- **`**kwds` - all other arguments are passed to the** surface function

OUTPUT: a 3d plot

EXAMPLES:

We plot a matrix that illustrates summation modulo n:

```
sage: n = 5; list_plot3d(matrix(RDF,n,[(i+j)%n for i in [1..n] for j in [1..n]])) # indirect doc
```

The interpolation type for matrices is 'linear'; for other types use other `list_plot3d()` input types.

We plot a matrix of values of sin:

```
sage: pi = float(pi)
sage: m = matrix(RDF, 6, [sin(i^2 + j^2) for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, texture='yellow', frame_aspect_ratio=[1,1,1/3]) # indirect doctest
sage: list_plot3d(m, texture='yellow', interpolation_type='linear') # indirect doctest
```

sage.plot.plot3d.list_plot3d.**list_plot3d_tuples**(*v*, *interpolation_type*, *texture*, ***kwds*)

A 3-dimensional plot of a surface defined by the list v of points in 3-dimensional space.

INPUT:

- `v` - something that defines a set of points in 3 space, for example:
  - a matrix

    This will be if using an interpolation type other than 'linear', or if using `num_points` with 'linear'; otherwise see `list_plot3d_matrix()`.
  - a list of 3-tuples
  - a list of lists (all of the same length, under same conditions as a matrix)
- `texture` - (default: "automatic", a solid light blue)

OPTIONAL KEYWORDS:

- `interpolation_type` - 'linear', 'nn' (nearest neighbor), 'spline'

  'linear' will perform linear interpolation

  The option 'nn' will interpolate by averaging the value of the nearest neighbors, this produces an interpolating function that is smoother than a linear interpolation, it has one derivative everywhere except at the sample points.

  The option 'spline' interpolates using a bivariate B-spline.

  When v is a matrix the default is to use linear interpolation, when v is a list of points the default is nearest neighbor.
- `degree` - an integer between 1 and 5, controls the degree of spline used for spline interpolation. For data that is highly oscillatory use higher values
- `point_list` - If point_list=True is passed, then if the array is a list of lists of length three, it will be treated as an array of points rather than a $3 \times n$ array.
- `num_points` - Number of points to sample interpolating function in each direction. By default for an $n \times n$ array this is n.
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot

EXAMPLES:

All of these use this function; see `list_plot3d()` for other list plots:

```
sage: pi = float(pi)
sage: m = matrix(RDF, 6, [sin(i^2 + j^2) for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, texture='yellow', interpolation_type='linear', num_points=5) # indirect doc
```

```
sage: list_plot3d(m, texture='yellow', interpolation_type='spline',frame_aspect_ratio=[1,1,1/3])
sage: show(list_plot3d([[1, 1, 1], [1, 2, 1], [0, 1, 3], [1, 0, 4]],point_list=True))
sage: list_plot3d([(1,2,3),(0,1,3),(2,1,4),(1,0,-2)], texture='yellow', num_points=50)
```

CHAPTER

EIGHT

PLOTTING FUNCTIONS

EXAMPLES:

```
sage: def f(x,y):
...       return math.sin(y*y+x*x)/math.sqrt(x*x+y*y+.0001)
...
sage: P = plot3d(f,(-3,3),(-3,3), adaptive=True, color=rainbow(60, 'rgbtuple'), max_bend=.1, max_dept
sage: P.show()

sage: def f(x,y):
...       return math.exp(x/5)*math.sin(y)
...
sage: P = plot3d(f,(-5,5),(-5,5), adaptive=True, color=['red','yellow'])
sage: from sage.plot.plot3d.plot3d import axes
sage: S = P + axes(6, color='black')
sage: S.show()
```

We plot "cape man":

```
sage: S = sphere(size=.5, color='yellow')

sage: from sage.plot.plot3d.shapes import Cone
sage: S += Cone(.5, .5, color='red').translate(0,0,.3)

sage: S += sphere((.45,-.1,.15), size=.1, color='white') + sphere((.51,-.1,.17), size=.05, color='bla
sage: S += sphere((.45, .1,.15),size=.1, color='white') + sphere((.51, .1,.17), size=.05, color='blac
sage: S += sphere((.5,0,-.2),size=.1, color='yellow')
sage: def f(x,y): return math.exp(x/5)*math.cos(y)
sage: P = plot3d(f,(-5,5),(-5,5), adaptive=True, color=['red','yellow'], max_depth=10)
sage: cape_man = P.scale(.2) + S.translate(1,0,0)
sage: cape_man.show(aspect_ratio=[1,1,1])
```

Or, we plot a very simple function indeed:

```
sage: plot3d(pi, (-1,1), (-1,1))
```

AUTHORS:

- Tom Boothby: adaptive refinement triangles
- Josh Kantor: adaptive refinement triangles
- Robert Bradshaw (2007-08): initial version of this file
- William Stein (2007-12, 2008-01): improving 3d plotting

- Oscar Lazo, William Cauchois, Jason Grout (2009-2010): Adding coordinate transformations

class sage.plot.plot3d.plot3d.**Cylindrical**(*dep_var*, *indep_vars*)
Bases: sage.plot.plot3d.plot3d._Coordinates

A cylindrical coordinate system for use with plot3d(transformation=...) where the position of a point is specified by three numbers:

•the *radial distance* (radius) from the z-axis

•the *azimuth angle* (azimuth) from the positive x-axis

•the *height* or *altitude* (height) above the xy-plane

These three variables must be specified in the constructor.

EXAMPLES:

Construct a cylindrical transformation for a function for height in terms of radius and azimuth:

```
sage: T = Cylindrical('height', ['radius', 'azimuth'])
```

If we construct some concrete variables, we can get a transformation:

```
sage: r, theta, z = var('r theta z')
sage: T.transform(radius=r, azimuth=theta, height=z)
(r*cos(theta), r*sin(theta), z)
```

We can plot with this transform. Remember that the dependent variable is the height, and the independent variables are the radius and the azimuth (in that order):

```
sage: plot3d(9-r^2, (r, 0, 3), (theta, 0, pi), transformation=T)
```

We next graph the function where the radius is constant:

```
sage: S=Cylindrical('radius', ['azimuth', 'height'])
sage: theta,z=var('theta, z')
sage: plot3d(3, (theta,0,2*pi), (z, -2, 2), transformation=S)
```

See also cylindrical_plot3d() for more examples of plotting in cylindrical coordinates.

transform(*radius=None*, *azimuth=None*, *height=None*)
A cylindrical coordinates transform.

EXAMPLE:

```
sage: T = Cylindrical('height', ['azimuth', 'radius'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), height=var('z'))
(r*cos(theta), r*sin(theta), z)
```

class sage.plot.plot3d.plot3d.**Spherical**(*dep_var*, *indep_vars*)
Bases: sage.plot.plot3d.plot3d._Coordinates

A spherical coordinate system for use with plot3d(transformation=...) where the position of a point is specified by three numbers:

•the *radial distance* (radius) from the origin

•the *azimuth angle* (azimuth) from the positive x-axis

•the *inclination angle* (inclination) from the positive z-axis

These three variables must be specified in the constructor.

EXAMPLES:

Construct a spherical transformation for a function for the radius in terms of the azimuth and inclination:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
```

If we construct some concrete variables, we can get a transformation in terms of those variables:

```
sage: r, phi, theta = var('r phi theta')
sage: T.transform(radius=r, azimuth=theta, inclination=phi)
(r*cos(theta)*sin(phi), r*sin(phi)*sin(theta), r*cos(phi))
```

We can plot with this transform. Remember that the dependent variable is the radius, and the independent variables are the azimuth and the inclination (in that order):

```
sage: plot3d(phi * theta, (theta, 0, pi), (phi, 0, 1), transformation=T)
```

We next graph the function where the inclination angle is constant:

```
sage: S=Spherical('inclination', ['radius', 'azimuth'])
sage: r,theta=var('r,theta')
sage: plot3d(3, (r,0,3), (theta, 0, 2*pi), transformation=S)
```

See also `spherical_plot3d()` for more examples of plotting in spherical coordinates.

transform(*radius=None*, *azimuth=None*, *inclination=None*)

A spherical coordinates transform.

EXAMPLE:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), inclination=var('phi'))
(r*cos(theta)*sin(phi), r*sin(phi)*sin(theta), r*cos(phi))
```

class sage.plot.plot3d.plot3d.**SphericalElevation**(*dep_var*, *indep_vars*)

Bases: sage.plot.plot3d.plot3d._Coordinates

A spherical coordinate system for use with `plot3d(transformation=...)` where the position of a point is specified by three numbers:

- the *radial distance* (`radius`) from the origin
- the *azimuth angle* (`azimuth`) from the positive x-axis
- the *elevation angle* (`elevation`) from the xy-plane toward the positive z-axis

These three variables must be specified in the constructor.

EXAMPLES:

Construct a spherical transformation for the radius in terms of the azimuth and elevation. Then, get a transformation in terms of those variables:

```
sage: T = SphericalElevation('radius', ['azimuth', 'elevation'])
sage: r, theta, phi = var('r theta phi')
sage: T.transform(radius=r, azimuth=theta, elevation=phi)
(r*cos(phi)*cos(theta), r*cos(phi)*sin(theta), r*sin(phi))
```

We can plot with this transform. Remember that the dependent variable is the radius, and the independent variables are the azimuth and the elevation (in that order):

```
sage: plot3d(phi * theta, (theta, 0, pi), (phi, 0, 1), transformation=T)
```

We next graph the function where the elevation angle is constant. This should be compared to the similar example for the `Spherical` coordinate system:

```
sage: SE=SphericalElevation('elevation', ['radius', 'azimuth'])
sage: r,theta=var('r,theta')
sage: plot3d(3, (r,0,3), (theta, 0, 2*pi), transformation=SE)
```

Plot a sin curve wrapped around the equator:

```
sage: P1=plot3d( (pi/12)*sin(8*theta), (r,0.99,1), (theta, 0, 2*pi), transformation=SE, plot_poi
sage: P2=sphere(center=(0,0,0), size=1, color='red', opacity=0.3)
sage: P1+P2
```

Now we graph several constant elevation functions alongside several constant inclination functions. This example illustrates the difference between the `Spherical` coordinate system and the `SphericalElevation` coordinate system:

```
sage: r, phi, theta = var('r phi theta')
sage: SE = SphericalElevation('elevation', ['radius', 'azimuth'])
sage: angles = [pi/18, pi/12, pi/6]
sage: P1 = [plot3d( a, (r,0,3), (theta, 0, 2*pi), transformation=SE, opacity=0.85, color='blue')

sage: S = Spherical('inclination', ['radius', 'azimuth'])
sage: P2 = [plot3d( a, (r,0,3), (theta, 0, 2*pi), transformation=S, opacity=0.85, color='red') f
sage: show(sum(P1+P2), aspect_ratio=1)
```

See also `spherical_plot3d()` for more examples of plotting in spherical coordinates.

transform(*radius=None*, *azimuth=None*, *elevation=None*)

A spherical elevation coordinates transform.

EXAMPLE:

```
sage: T = SphericalElevation('radius', ['azimuth', 'elevation'])
sage: T.transform(radius=var('r'), azimuth=var('theta'), elevation=var('phi'))
(r*cos(phi)*cos(theta), r*cos(phi)*sin(theta), r*sin(phi))
```

class sage.plot.plot3d.plot3d.**TrivialTriangleFactory**

Class emulating behavior of `TriangleFactory` but simply returning a list of vertices for both regular and smooth triangles.

smooth_triangle(*a*, *b*, *c*, *da*, *db*, *dc*, *color=None*)

Function emulating behavior of `smooth_triangle()` but simply returning a list of vertices.

INPUT:

- `a`, `b`, `c` : triples (x,y,z) representing corners on a triangle in 3-space
- `da`, `db`, `dc` : ignored
- `color` : ignored

OUTPUT:

- the list `[a,b,c]`

TESTS:

```
sage: from sage.plot.plot3d.plot3d import TrivialTriangleFactory
sage: factory = TrivialTriangleFactory()
sage: sm_tri = factory.smooth_triangle([0,0,0],[0,0,1],[1,1,0],[0,0,1],[0,2,0],[1,0,0])
sage: sm_tri
[[0, 0, 0], [0, 0, 1], [1, 1, 0]]
```

triangle(*a*, *b*, *c*, *color=None*)

Function emulating behavior of `triangle()` but simply returning a list of vertices.

INPUT:

- `a, b, c` : triples (x,y,z) representing corners on a triangle in 3-space
- `color`: ignored

OUTPUT:

- the list `[a,b,c]`

TESTS:

```
sage: from sage.plot.plot3d.plot3d import TrivialTriangleFactory
sage: factory = TrivialTriangleFactory()
sage: tri = factory.triangle([0,0,0],[0,0,1],[1,1,0])
sage: tri
[[0, 0, 0], [0, 0, 1], [1, 1, 0]]
```

sage.plot.plot3d.plot3d.**axes**(*scale=1*, *radius=None*, ***kwds*)

Creates basic axes in three dimensions. Each axis is a three dimensional arrow object.

INPUT:

- `scale` - (default: 1) The length of the axes (all three will be the same).
- `radius` - (default: .01) The radius of the axes as arrows.

EXAMPLES:

```
sage: from sage.plot.plot3d.plot3d import axes
sage: S = axes(6, color='black'); S

sage: T = axes(2, .5); T
```

sage.plot.plot3d.plot3d.**cylindrical_plot3d**(*f*, *urange*, *vrange*, ***kwds*)

Plots a function in cylindrical coordinates. This function is equivalent to:

```
sage: r,u,v=var('r,u,v')
sage: f=u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: T = (r*cos(u), r*sin(u), v, [u,v])
sage: plot3d(f, urange, vrange, transformation=T)
```

or equivalently:

```
sage: T = Cylindrical('radius', ['azimuth', 'height'])
sage: f=lambda u,v: u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: plot3d(f, urange, vrange, transformation=T)
```

INPUT:

- `f` - a symbolic expression or function of two variables, representing the radius from the z-axis.
- `urange` - a 3-tuple (u, u_min, u_max), the domain of the azimuth variable.
- `vrange` - a 3-tuple (v, v_min, v_max), the domain of the elevation (z) variable.

EXAMPLES:

A portion of a cylinder of radius 2:

```
sage: theta,z=var('theta,z')
sage: cylindrical_plot3d(2,(theta,0,3*pi/2),(z,-2,2))
```

Some random figures:

```
sage: cylindrical_plot3d(cosh(z),(theta,0,2*pi),(z,-2,2))

sage: cylindrical_plot3d(e^(-z^2)*(cos(4*theta)+2)+1,(theta,0,2*pi),(z,-2,2),plot_points=[80,80]
```

sage.plot.plot3d.plot3d.**plot3d**(*f*, *urange*, *vrange*, *adaptive=False*, *transformation=None*, ***kwds*)

INPUT:

- `f` - a symbolic expression or function of 2 variables
- `urange` - a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- `vrange` - a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)
- `adaptive` - (default: False) whether to use adaptive refinement to draw the plot (slower, but may look better). This option does NOT work in conjuction with a transformation (see below).
- `mesh` - bool (default: False) whether to display mesh grid lines
- `dots` - bool (default: False) whether to display dots at mesh grid points
- `plot_points` - (default: "automatic") initial number of sample points in each direction; an integer or a pair of integers
- `transformation` - (default: None) a transformation to apply. May be a 3 or 4-tuple (x_func, y_func, z_func, independent_vars) where the first 3 items indicate a transformation to cartesian coordinates (from your coordinate system) in terms of u, v, and the function variable fvar (for which the value of f will be substituted). If a 3-tuple is specified, the independent variables are chosen from the range variables. If a 4-tuple is specified, the 4th element is a list of independent variables. `transformation` may also be a predefined coordinate system transformation like Spherical or Cylindrical.

Note: `mesh` and `dots` are not supported when using the Tachyon raytracer renderer.

EXAMPLES: We plot a 3d function defined as a Python function:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2))
```

We plot the same 3d function but using adaptive refinement:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True)
```

Adaptive refinement but with more points:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True, initial_depth=5)
```

We plot some 3d symbolic functions:

```
sage: var('x,y')
(x, y)
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
sage: plot3d(sin(x*y), (x, -pi, pi), (y, -pi, pi))
```

We give a plot with extra sample points:

```
sage: var('x,y')
(x, y)
sage: plot3d(sin(x^2+y^2),(x,-5,5),(y,-5,5), plot_points=200)
sage: plot3d(sin(x^2+y^2),(x,-5,5),(y,-5,5), plot_points=[10,100])
```

A 3d plot with a mesh:

```
sage: var('x,y')
(x, y)
sage: plot3d(sin(x-y)*y*cos(x),(x,-3,3),(y,-3,3), mesh=True)
```

Two wobby translucent planes:

```
sage: x,y = var('x,y')
sage: P = plot3d(x+y+sin(x*y), (x,-10,10),(y,-10,10), opacity=0.87, color='blue')
sage: Q = plot3d(x-2*y-cos(x*y),(x,-10,10),(y,-10,10),opacity=0.3,color='red')
sage: P + Q
```

We draw two parametric surfaces and a transparent plane:

```
sage: L = plot3d(lambda x,y: 0, (-5,5), (-5,5), color="lightblue", opacity=0.8)
sage: P = plot3d(lambda x,y: 4 - x^3 - y^2, (-2,2), (-2,2), color='green')
sage: Q = plot3d(lambda x,y: x^3 + y^2 - 4, (-2,2), (-2,2), color='orange')
sage: L + P + Q
```

We draw the "Sinus" function (water ripple-like surface):

```
sage: x, y = var('x y')
sage: plot3d(sin(pi*(x^2+y^2))/2,(x,-1,1),(y,-1,1))
```

Hill and valley (flat surface with a bump and a dent):

```
sage: x, y = var('x y')
sage: plot3d( 4*x*exp(-x^2-y^2), (x,-2,2), (y,-2,2))
```

An example of a transformation:

```
sage: r, phi, z = var('r phi z')
sage: trans=(r*cos(phi),r*sin(phi),z)
sage: plot3d(cos(r),(r,0,17*pi/2),(phi,0,2*pi),transformation=trans,opacity=0.87).show(aspect_ra
```

An example of a transformation with symbolic vector:

```
sage: cylindrical(r,theta,z)=[r*cos(theta),r*sin(theta),z]
sage: plot3d(3,(theta,0,pi/2),(z,0,pi/2),transformation=cylindrical)
```

Many more examples of transformations:

```
sage: u, v, w = var('u v w')
sage: rectangular=(u,v,w)
sage: spherical=(w*cos(u)*sin(v),w*sin(u)*sin(v),w*cos(v))
sage: cylindric_radial=(w*cos(u),w*sin(u),v)
sage: cylindric_axial=(v*cos(u),v*sin(u),w)
sage: parabolic_cylindrical=(w*v,(v^2-w^2)/2,u)
```

Plot a constant function of each of these to get an idea of what it does:

```
sage: A = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=rectangular,plot_points=[100,100])
sage: B = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=spherical,plot_points=[100,100])
sage: C = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=cylindric_radial,plot_points=[100,100])
sage: D = plot3d(2,(u,-pi,pi),(v,0,pi),transformation=cylindric_axial,plot_points=[100,100])
sage: E = plot3d(2,(u,-pi,pi),(v,-pi,pi),transformation=parabolic_cylindrical,plot_points=[100,1
sage: @interact
... def _(which_plot=[A,B,C,D,E]):
...       show(which_plot)
<html>...
```

Now plot a function:

```
sage: g=3+sin(4*u)/2+cos(4*v)/2
sage: F = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=rectangular,plot_points=[100,100])
sage: G = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=spherical,plot_points=[100,100])
sage: H = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=cylindric_radial,plot_points=[100,100])
sage: I = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=cylindric_axial,plot_points=[100,100])
sage: J = plot3d(g,(u,-pi,pi),(v,0,pi),transformation=parabolic_cylindrical,plot_points=[100,100
sage: @interact
... def _(which_plot=[F, G, H, I, J]):
...       show(which_plot)
<html>...
```

TESTS:

Make sure the transformation plots work:

```
sage: show(A + B + C + D + E)
sage: show(F + G + H + I + J)
```

Listing the same plot variable twice gives an error:

```
sage: x, y = var('x y')
sage: plot3d( 4*x*exp(-x^2-y^2), (x,-2,2), (x,-2,2))
Traceback (most recent call last):
...
ValueError: range variables should be distinct, but there are duplicates
```

sage.plot.plot3d.plot3d.**plot3d_adaptive**(*f*, *x_range*, *y_range*, *color='automatic'*, *grad_f=None*, *max_bend=0.5*, *max_depth=5*, *initial_depth=4*, *num_colors=128*, ***kwds*)

Adaptive 3d plotting of a function of two variables.

This is used internally by the plot3d command when the option `adaptive=True` is given.

INPUT:

- `f` - a symbolic function or a Python function of 3 variables.
- `x_range` - x range of values: 2-tuple (xmin, xmax) or 3-tuple (x,xmin,xmax)
- `y_range` - y range of values: 2-tuple (ymin, ymax) or 3-tuple (y,ymin,ymax)
- `grad_f` - gradient of f as a Python function
- `color` - "automatic" - a rainbow of num_colors colors
- `num_colors` - (default: 128) number of colors to use with default color
- `max_bend` - (default: 0.5)
- `max_depth` - (default: 5)
- `initial_depth` - (default: 4)
- `**kwds` - standard graphics parameters

EXAMPLES:

We plot $\sin(xy)$:

```
sage: from sage.plot.plot3d.plot3d import plot3d_adaptive
sage: x,y=var('x,y'); plot3d_adaptive(sin(x*y), (x,-pi,pi), (y,-pi,pi), initial_depth=5)
```

sage.plot.plot3d.plot3d.**spherical_plot3d**(*f*, *urange*, *vrange*, ***kwds*)

Plots a function in spherical coordinates. This function is equivalent to:

```
sage: r,u,v=var('r,u,v')
sage: f=u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: T = (r*cos(u)*sin(v), r*sin(u)*sin(v), r*cos(v), [u,v])
sage: plot3d(f, urange, vrange, transformation=T)
```

or equivalently:

```
sage: T = Spherical('radius', ['azimuth', 'inclination'])
sage: f=lambda u,v: u*v; urange=(u,0,pi); vrange=(v,0,pi)
sage: plot3d(f, urange, vrange, transformation=T)
```

INPUT:

- `f` - a symbolic expression or function of two variables.
- `urange` - a 3-tuple (u, u_min, u_max), the domain of the azimuth variable.
- `vrange` - a 3-tuple (v, v_min, v_max), the domain of the inclination variable.

EXAMPLES:

A sphere of radius 2:

```
sage: x,y=var('x,y')
sage: spherical_plot3d(2,(x,0,2*pi),(y,0,pi))
```

The real and imaginary parts of a spherical harmonic with $l = 2$ and $m = 1$:

```
sage: phi, theta = var('phi, theta')
sage: Y = spherical_harmonic(2, 1, theta, phi)
sage: rea = spherical_plot3d(abs(real(Y)), (phi,0,2*pi), (theta,0,pi), color='blue', opacity=0.6
sage: ima = spherical_plot3d(abs(imag(Y)), (phi,0,2*pi), (theta,0,pi), color='red', opacity=0.6)
sage: (rea + ima).show(aspect_ratio=1)  # long time (4s on sage.math, 2011)
```

A drop of water:

```
sage: x,y=var('x,y')
sage: spherical_plot3d(e^-y,(x,0,2*pi),(y,0,pi),opacity=0.5).show(frame=False)
```

An object similar to a heart:

```
sage: x,y=var('x,y')
sage: spherical_plot3d((2+cos(2*x))*(y+1),(x,0,2*pi),(y,0,pi),rgbcolor=(1,.1,.1))
```

Some random figures:

```
sage: x,y=var('x,y')
sage: spherical_plot3d(1+sin(5*x)/5,(x,0,2*pi),(y,0,pi),rgbcolor=(1,0.5,0),plot_points=(80,80),c
```

```
sage: x,y=var('x,y')
sage: spherical_plot3d(1+2*cos(2*y),(x,0,3*pi/2),(y,0,pi)).show(aspect_ratio=(1,1,1))
```

CHAPTER

NINE

PLATONIC SOLIDS

EXAMPLES: The five platonic solids in a row;

```
sage: G = tetrahedron((0,-3.5,0), color='blue') + cube((0,-2,0),color=(.25,0,.5)) +\
      octahedron(color='red') + dodecahedron((0,2,0), color='orange') +\
      icosahedron(center=(0,4,0), color='yellow')
sage: G.show(aspect_ratio=[1,1,1])
```

All the platonic solids in the same place:

```
sage: G = tetrahedron(color='blue',opacity=0.7) + \
      cube(color=(.25,0,.5), opacity=0.7) +\
      octahedron(color='red', opacity=0.7) + \
      dodecahedron(color='orange', opacity=0.7) + icosahedron(opacity=0.7)
sage: G.show(aspect_ratio=[1,1,1])
```

Display nice faces only:

```
sage: icosahedron().stickers(['red','blue'], .075, .1)
```

AUTHORS:

- Robert Bradshaw (2007, 2008): initial version
- William Stein

sage.plot.plot3d.platonic.**cube**(*center=(0, 0, 0)*, *size=1*, *color=None*, *frame_thickness=0*, *frame_color=None*, ***kwds*)

A 3D cube centered at the origin with default side lengths 1.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1) the side lengths of the cube
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `frame_thickness` - (default: 0) if positive, then thickness of the frame
- `frame_color` - (default: None) if given, gives the color of the frame
- `opacity` - (default: 1) if less than 1 then it's transparent

EXAMPLES:

A simple cube:

```
sage: cube()
```

A red cube:

```
sage: cube(color="red")
```

A transparent grey cube that contains a red cube:

```
sage: cube(opacity=0.8, color='grey') + cube(size=3/4)
```

A transparent colored cube:

```
sage: cube(color=['red', 'green', 'blue'], opacity=0.5)
```

A bunch of random cubes:

```
sage: v = [(random(), random(), random()) for _ in [1..30]]
sage: sum([cube((10*a,10*b,10*c), size=random()/3, color=(a,b,c)) for a,b,c in v])
```

Non-square cubes (boxes):

```
sage: cube(aspect_ratio=[1,1,1]).scale([1,2,3])
sage: cube(color=['red', 'blue', 'green'],aspect_ratio=[1,1,1]).scale([1,2,3])
```

And one that is colored:

```
sage: cube(color=['red', 'blue', 'green', 'black', 'white', 'orange'],                    aspect_
```

A nice translucent color cube with a frame:

```
sage: c = cube(color=['red', 'blue', 'green'], frame=False, frame_thickness=2,
sage: c
```

A raytraced color cube with frame and transparency:

```
sage: c.show(viewer='tachyon')
```

This shows #11272 has been fixed:

```
sage: cube(center=(10, 10, 10), size=0.5).bounding_box()
((9.75, 9.75, 9.75), (10.25, 10.25, 10.25))
```

AUTHORS:

- William Stein

sage.plot.plot3d.platonic.**dodecahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

A dodecahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES: A plain Dodecahedron:

```
sage: dodecahedron()
```

A translucent dodecahedron that contains a black sphere:

```
sage: dodecahedron(color='orange', opacity=0.8) + \
      sphere(size=0.5, color='black')
```

CONSTRUCTION: This is how we construct a dodecahedron. We let one point be $Q = (0, 1, 0)$.

Now there are three points spaced equally on a circle around the north pole. The other requirement is that the angle between them be the angle of a pentagon, namely $3\pi/5$. This is enough to determine them. Placing one on the xz-plane we have.

$P_1 = \left(t, 0, \sqrt{1-t^2}\right)$

$P_2 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$

$P_3 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$

Solving $\frac{(P_1-Q)\cdot(P_2-Q)}{|P_1-Q||P_2-Q|} = \cos(3\pi/5)$ we get $t = 2/3$.

Now we have 6 points $R_1, ..., R_6$ to close the three top pentagons. These can be found by mirroring P_2 and P_3 by the yz-plane and rotating around the y-axis by the angle θ from Q to P_1. Note that $\cos(\theta) = t = 2/3$ and so $\sin(\theta) = \sqrt{5}/3$. Rotation gives us the other four.

Now we reflect through the origin for the bottom half.

AUTHORS:

- Robert Bradshaw, William Stein

sage.plot.plot3d.platonic.**icosahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

An icosahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: icosahedron()
```

Two icosahedrons at different positions of different sizes.

```
sage: icosahedron((-1/2,0,1), color='orange') + \
      icosahedron((2,0,1), size=1/2, aspect_ratio=[1,1,1])
```

sage.plot.plot3d.platonic.**index_face_set**(*face_list*, *point_list*, *enclosed*, ***kwds*)

Helper function that creates `IndexFaceSet` object for the tetrahedron, dodecahedron, and icosahedron.

INPUT:

- `face_list` - list of faces, given explicitly from the solid invocation
- `point_list` - list of points, given explicitly from the solid invocation
- `enclosed` - boolean (default passed is always True for these solids)

TESTS:

Verify that these are working and passing on keywords:

```
sage: tetrahedron(center=(2,0,0),size=2,color='red')

sage: dodecahedron(center=(2,0,0),size=2,color='red')

sage: icosahedron(center=(2,0,0),size=2,color='red')
```

sage.plot.plot3d.platonic.**octahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

Return an octahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: octahedron((1,4,3), color='orange') + \
            octahedron((0,2,1), size=2, opacity=0.6)
```

sage.plot.plot3d.platonic.**prep**(*G*, *center*, *size*, *kwds*)

Helper function that scales and translates the platonic solid, and passes extra keywords on.

INPUT:

- `center` - 3-tuple indicating the center (default passed from `index_face_set()` is the origin $(0, 0, 0)$)
- `size` - number indicating amount to scale by (default passed from `index_face_set()` is 1)
- `kwds` - a dictionary of keywords, passed from solid invocation by `index_face_set()`

TESTS:

Verify that scaling and moving the center work together properly, and that keywords are passed (see Trac #10796):

```
sage: octahedron(center=(2,0,0),size=2,color='red')
```

sage.plot.plot3d.platonic.**tetrahedron**(*center=(0, 0, 0)*, *size=1*, ***kwds*)

A 3d tetrahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a word that describes a color
- `rgbcolor` - (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES: A default colored tetrahedron at the origin:

```
sage: tetrahedron()
```

A transparent green tetrahedron in front of a solid red one:

```
sage: tetrahedron(opacity=0.8, color='green') + tetrahedron((-2,1,0),color='red')
```

A translucent tetrahedron sharing space with a sphere:

```
sage: tetrahedron(color='yellow',opacity=0.7) + sphere(r=.5, color='red')
```

A big tetrahedron:

```
sage: tetrahedron(size=10)
```

A wide tetrahedron:

```
sage: tetrahedron(aspect_ratio=[1,1,1]).scale((4,4,1))
```

A red and blue tetrahedron touching noses:

```
sage: tetrahedron(color='red') + tetrahedron((0,0,-2)).scale([1,1,-1])
```

A Dodecahedral complex of 5 tetrahedrons (a more elaborate examples from Peter Jipsen):

```
sage: v=(sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t=acos(sqrt(5.)/3)/2
sage: t1=tetrahedron(aspect_ratio=(1,1,1), opacity=0.5).rotateZ(t)
sage: t2=tetrahedron(color='red', opacity=0.5).rotateZ(t).rotate(v,2*pi/5)
sage: t3=tetrahedron(color='green', opacity=0.5).rotateZ(t).rotate(v,4*pi/5)
sage: t4=tetrahedron(color='yellow', opacity=0.5).rotateZ(t).rotate(v,6*pi/5)
sage: t5=tetrahedron(color='orange', opacity=0.5).rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False, zoom=1.3)
```

AUTHORS:

- Robert Bradshaw and William Stein

CHAPTER

TEN

CLASSES FOR LINES, FRAMES, RULERS, SPHERES, POINTS, DOTS, AND TEXT

AUTHORS:

- William Stein (2007-12): initial version
- William Stein and Robert Bradshaw (2008-01): Many improvements

class sage.plot.plot3d.shapes2.**Line**(*points*, *thickness=5*, *corner_cutoff=0.5*, *arrow_head=False*, ***kwds*)

Bases: sage.plot.plot3d.base.PrimitiveObject

Draw a 3d line joining a sequence of points.

This line has a fixed diameter unaffected by transformations and zooming. It may be smoothed if `corner_cutoff < 1`.

INPUT:

- `points` - list of points to pass through
- `thickness` - diameter of the line
- `corner_cutoff` - threshold for smoothing (see the corners() method) this is the minimum cosine between adjacent segments to smooth
- `arrow_head` - if True make this curve into an arrow

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(i*math.sin(i), i*math.cos(i), i/3) for i in range(30)], arrow_head=True)
```

Smooth angles less than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0)
```

bounding_box()

Returns the lower and upper corners of a 3-D bounding box for `self`. This is used for rendering and `self` should fit entirely within this box. In this case, we return the highest and lowest values of each coordinate among all points.

TESTS:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: L = Line([(i,i^2-1,-2*ln(i)) for i in [10,20,30]])
sage: L.bounding_box()
((10.0, 99.0, -6.802394763324311), (30.0, 899.0, -4.605170185988092))
```

corners (*corner_cutoff=None*, *max_len=None*)

Figures out where the curve turns too sharply to pretend it's smooth.

INPUT: Maximum cosine of angle between adjacent line segments before adding a corner

OUTPUT: List of points at which to start a new line. This always includes the first point, and never the last.

EXAMPLES:

Every point:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=1).corners()
[(0, 0, 0), (1, 0, 0), (2, 1, 0)]
```

Greater than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0).corners()
[(0, 0, 0), (2, 1, 0)]
```

No corners:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=-1).corners()
(0, 0, 0)
```

An intermediate value:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=.5).corners()
[(0, 0, 0), (2, 1, 0)]
```

jmol_repr (*render_params*)

Returns representation of the object suitable for plotting using Jmol.

TESTS:

```
sage: L = line3d([(cos(i),sin(i),i^2) for i in srange(0,10,.01)],color='red')
sage: L.jmol_repr(L.default_render_params())[0][:42]
'draw line_1 diameter 1 curve {1.0 0.0 0.0}'
```

obj_repr (*render_params*)

Returns complete representation of the line as an object.

TESTS:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: L = Line([(cos(i),sin(i),i^2) for i in srange(0,10,.01)],color='red')
sage: L.obj_repr(L.default_render_params())[0][0][0][2][:3]
['v 0.99995 0.00999983 0.0001', 'v 1.00007 0.0102504 -0.0248984', 'v 1.02376 0.010195 -0.007
```

tachyon_repr (*render_params*)

Returns representation of the line suitable for plotting using the Tachyon ray tracer.

TESTS:

```
sage: L = line3d([(cos(i),sin(i),i^2) for i in srange(0,10,.01)],color='red')
sage: L.tachyon_repr(L.default_render_params())[0]
'FCylinder base 1.0 0.0 0.0 apex 0.999950000417 0.00999983333417 0.0001 rad 0.005 texture...
```

class sage.plot.plot3d.shapes2.**Point**(*center*, *size=1*, ***kwds*)
Bases: sage.plot.plot3d.base.PrimitiveObject

Create a position in 3-space, represented by a sphere of fixed size.

INPUT:

- center - point (3-tuple)
- size - (default: 1)

EXAMPLE:

We normally access this via the point3d function. Note that extra keywords are correctly used:

```
sage: point3d((4,3,2),size=2,color='red',opacity=.5)
```

bounding_box()
Returns the lower and upper corners of a 3-D bounding box for self. This is used for rendering and self should fit entirely within this box. In this case, we simply return the center of the point.

TESTS:

```
sage: P = point3d((-3,2,10),size=7)
sage: P.bounding_box()
((-3.0, 2.0, 10.0), (-3.0, 2.0, 10.0))
```

jmol_repr(*render_params*)
Returns representation of the object suitable for plotting using Jmol.

TESTS:

```
sage: P = point3d((1,2,3),size=3,color='purple')
sage: P.jmol_repr(P.default_render_params())
['draw point_1 DIAMETER 3 {1.0 2.0 3.0}\ncolor $point_1  [128,0,128]']
```

obj_repr(*render_params*)
Returns complete representation of the point as a sphere.

TESTS:

```
sage: P = point3d((1,2,3),size=3,color='purple')
sage: P.obj_repr(P.default_render_params())[0][0:2]
['g obj_1', 'usemtl texture...']
```

tachyon_repr(*render_params*)
Returns representation of the point suitable for plotting using the Tachyon ray tracer.

TESTS:

```
sage: P = point3d((1,2,3),size=3,color='purple')
sage: P.tachyon_repr(P.default_render_params())
'Sphere center 1.0 2.0 3.0 Rad 0.015 texture...'
```

sage.plot.plot3d.shapes2.**bezier3d**(*path*, *aspect_ratio=[1, 1, 1]*, *color='blue'*, *opacity=1*, *thickness=2*, ***options*)
Draws a 3-dimensional bezier path. Input is similar to bezier_path, but each point in the path and each control point is required to have 3 coordinates.

INPUT:

- **path - a list of curves, which each is a list of points. See further** detail below.
- thickness - (default: 2)

- `color` - a word that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent
- `aspect_ratio` - (default:[1,1,1])

The path is a list of curves, and each curve is a list of points. Each point is a tuple (x,y,z).

The first curve contains the endpoints as the first and last point in the list. All other curves assume a starting point given by the last entry in the preceding list, and take the last point in the list as their opposite endpoint. A curve can have 0, 1 or 2 control points listed between the endpoints. In the input example for path below, the first and second curves have 2 control points, the third has one, and the fourth has no control points:

```
path = [[p1, c1, c2, p2], [c3, c4, p3], [c5, p4], [p5], ...]
```

In the case of no control points, a straight line will be drawn between the two endpoints. If one control point is supplied, then the curve at each of the endpoints will be tangent to the line from that endpoint to the control point. Similarly, in the case of two control points, at each endpoint the curve will be tangent to the line connecting that endpoint with the control point immediately after or immediately preceding it in the list.

So in our example above, the curve between p1 and p2 is tangent to the line through p1 and c1 at p1, and tangent to the line through p2 and c2 at p2. Similarly, the curve between p2 and p3 is tangent to line(p2,c3) at p2 and tangent to line(p3,c4) at p3. Curve(p3,p4) is tangent to line(p3,c5) at p3 and tangent to line(p4,c5) at p4. Curve(p4,p5) is a straight line.

EXAMPLES:

```
sage: path = [[(0,0,0),(.5,.1,.2),(.75,3,-1),(1,1,0)],[(.5,1,.2),(1,.5,0)],[(.7,.2,.5)]]
sage: b = bezier3d(path, color='green')
sage: b
```

To construct a simple curve, create a list containing a single list:

```
sage: path = [[(0,0,0),(1,0,0),(0,1,0),(0,1,1)]]
sage: curve = bezier3d(path, thickness=5, color='blue')
sage: curve
```

sage.plot.plot3d.shapes2.**frame3d**(*lower_left*, *upper_right*, ***kwds*)

Draw a frame in 3-D. Primarily used as a helper function for creating frames for 3-D graphics viewing.

INPUT:

- `lower_left` - the lower left corner of the frame, as a list, tuple, or vector.
- `upper_right` - the upper right corner of the frame, as a list, tuple, or vector.

Type `line3d.options` for a dictionary of the default options for lines, which are also available.

EXAMPLES:

A frame:

```
sage: from sage.plot.plot3d.shapes2 import frame3d
sage: frame3d([1,3,2],vector([2,5,4]),color='red')
```

This is usually used for making an actual plot:

```
sage: y = var('y')
sage: plot3d(sin(x^2+y^2),(x,0,pi),(y,0,pi))
```

sage.plot.plot3d.shapes2.**frame_labels**(*lower_left*, *upper_right*, *label_lower_left*, *label_upper_right*, *eps=1*, ***kwds*)

Draw correct labels for a given frame in 3-D. Primarily used as a helper function for creating frames for 3-D graphics viewing - do not use directly unless you know what you are doing!

INPUT:

- `lower_left` - the lower left corner of the frame, as a list, tuple, or vector.
- `upper_right` - the upper right corner of the frame, as a list, tuple, or vector.
- `label_lower_left` - the label for the lower left corner of the frame, as a list, tuple, or vector. This label must actually have all coordinates less than the coordinates of the other label.
- `label_upper_right` - the label for the upper right corner of the frame, as a list, tuple, or vector. This label must actually have all coordinates greater than the coordinates of the other label.
- `eps` - (default: 1) a parameter for how far away from the frame to put the labels.

Type `line3d.options` for a dictionary of the default options for lines, which are also available.

EXAMPLES:

We can use it directly:

```
sage: from sage.plot.plot3d.shapes2 import frame_labels
sage: frame_labels([1,2,3],[4,5,6],[1,2,3],[4,5,6])
```

This is usually used for making an actual plot:

```
sage: y = var('y')
sage: P = plot3d(sin(x^2+y^2),(x,0,pi),(y,0,pi))
sage: a,b = P._rescale_for_frame_aspect_ratio_and_zoom(1.0,[1,1,1],1)
sage: F = frame_labels(a,b,*P._box_for_aspect_ratio("automatic",a,b))
sage: F.jmol_repr(F.default_render_params())[0]
[['select atomno = 1', 'color atom  [76,76,76]', 'label "0.0"']]
```

TESTS:

```
sage: frame_labels([1,2,3],[4,5,6],[1,2,3],[1,3,4])
Traceback (most recent call last):
...
ValueError: Ensure the upper right labels are above and to the right of the lower left labels.
```

sage.plot.plot3d.shapes2.**line3d**(*points*, *thickness=1*, *radius=None*, *arrow_head=False*, ***kwds*)

Draw a 3d line joining a sequence of points.

One may specify either a thickness or radius. If a thickness is specified, this line will have a constant diameter regardless of scaling and zooming. If a radius is specified, it will behave as a series of cylinders.

INPUT:

- `points` - a list of at least 2 points
- `thickness` - (default: 1)
- `radius` - (default: None)
- `arrow_head` - (default: False)
- `color` - a word that describes a color
- `rgbcolor` - (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

A line in 3-space:

```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)])
```

The same line but red:

```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)], color='red')
```

The points of the line provided as a numpy array:

```
sage: import numpy
sage: line3d(numpy.array([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)]))
```

A transparent thick green line and a little blue line:

```
sage: line3d([(0,0,0), (1,1,1), (1,0,2)], opacity=0.5, radius=0.1, \
            color='green') + line3d([(0,1,0), (1,0,2)])
```

A Dodecahedral complex of 5 tetrahedrons (a more elaborate examples from Peter Jipsen):

```
sage: def tetra(col):
...       return line3d([(0,0,1), (2*sqrt(2.)/3,0,-1./3), (-sqrt(2.)/3, sqrt(6.)/3,-1./3),\
...              (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (0,0,1), (-sqrt(2.)/3, sqrt(6.)/3,-1./3),\
...              (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (2*sqrt(2.)/3,0,-1./3)],\
...              color=col, thickness=10, aspect_ratio=[1,1,1])
...
sage: v  = (sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t  = acos(sqrt(5.)/3)/2
sage: t1 = tetra('blue').rotateZ(t)
sage: t2 = tetra('red').rotateZ(t).rotate(v,2*pi/5)
sage: t3 = tetra('green').rotateZ(t).rotate(v,4*pi/5)
sage: t4 = tetra('yellow').rotateZ(t).rotate(v,6*pi/5)
sage: t5 = tetra('orange').rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False)
```

TESTS:

Copies are made of the input list, so the input list does not change:

```
sage: mypoints = [vector([1,2,3]), vector([4,5,6])]
sage: type(mypoints[0])
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: L = line3d(mypoints)
sage: type(mypoints[0])
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

The copies are converted to a list, so we can pass in immutable objects too:

```
sage: L = line3d(((0,0,0),(1,2,3)))
```

This function should work for anything than can be turned into a list, such as iterators and such (see ticket #10478):

```
sage: line3d(iter([(0,0,0), (sqrt(3), 2, 4)]))
sage: line3d((x, x^2, x^3) for x in range(5))
sage: from itertools import izip; line3d(izip([2,3,5,7], [11, 13, 17, 19], [-1, -2, -3, -4]))
```

sage.plot.plot3d.shapes2.**point3d**(*v*, *size=5*, ***kwds*)

Plot a point or list of points in 3d space.

INPUT:

- v – a point or list of points

- `size` – (default: 5) size of the point (or points)
- `color` – a word that describes a color
- `rgbcolor` – (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` – (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: sum([point3d((i,i^2,i^3), size=5) for i in range(10)])
```

We check to make sure this works with vectors and other iterables:

```
sage: pl = point3d([vector(ZZ,(1, 0, 0)), vector(ZZ,(0, 1, 0)), (-1, -1, 0)])
sage: print point(vector((2,3,4)))
Graphics3d Object

sage: c = polytopes.n_cube(3)
sage: v = c.vertices()[0];  v
A vertex at (-1, -1, -1)
sage: print point(v)
Graphics3d Object
```

We check to make sure the options work:

```
sage: point3d((4,3,2),size=20,color='red',opacity=.5)
```

numpy arrays can be provided as input:

```
sage: import numpy
sage: point3d(numpy.array([1,2,3]))

sage: point3d(numpy.array([[1,2,3], [4,5,6], [7,8,9]]))
```

sage.plot.plot3d.shapes2.**polygon3d**(*points*, *color=(0, 0, 1)*, *opacity=1*, ***options*)

Draw a polygon in 3d.

INPUT:

- `points` - the vertices of the polygon

Type `polygon3d.options` for a dictionary of the default options for polygons. You can change this to change the defaults for all future polygons. Use `polygon3d.reset()` to reset to the default options.

EXAMPLES:

A simple triangle:

```
sage: polygon3d([[0,0,0], [1,2,3], [3,0,0]])
```

Some modern art – a random polygon:

```
sage: v = [(randrange(-5,5), randrange(-5,5), randrange(-5, 5)) for _ in range(10)]
sage: polygon3d(v)
```

A bent transparent green triangle:

```
sage: polygon3d([[1, 2, 3], [0,1,0], [1,0,1], [3,0,0]], color=(0,1,0), alpha=0.7)
```

sage.plot.plot3d.shapes2.**ruler**(*start*, *end*, *ticks=4*, *sub_ticks=4*, *absolute=False*, *snap=False*, ***kwds*)

Draw a ruler in 3-D, with major and minor ticks.

INPUT:

- start - the beginning of the ruler, as a list, tuple, or vector.
- end - the end of the ruler, as a list, tuple, or vector.
- ticks - (default: 4) the number of major ticks shown on the ruler.
- sub_ticks - (default: 4) the number of shown subdivisions between each major tick.
- absolute - (default: False) if True, makes a huge ruler in the direction of an axis.
- snap - (default: False) if True, snaps to an implied grid.

Type line3d.options for a dictionary of the default options for lines, which are also available.

EXAMPLES:

A ruler:

```
sage: from sage.plot.plot3d.shapes2 import ruler
sage: R = ruler([1,2,3],vector([2,3,4])); R
```

A ruler with some options:

```
sage: R = ruler([1,2,3],vector([2,3,4]),ticks=6, sub_ticks=2, color='red'); R
```

The keyword snap makes the ticks not necessarily coincide with the ruler:

```
sage: ruler([1,2,3],vector([1,2,4]),snap=True)
```

The keyword absolute makes a huge ruler in one of the axis directions:

```
sage: ruler([1,2,3],vector([1,2,4]),absolute=True)
```

TESTS:

```
sage: ruler([1,2,3],vector([1,3,4]),absolute=True)
Traceback (most recent call last):
...
ValueError: Absolute rulers only valid for axis-aligned paths
```

sage.plot.plot3d.shapes2.**ruler_frame**(*lower_left*, *upper_right*, *ticks=4*, *sub_ticks=4*, ***kwds*)

Draw a frame made of 3-D rulers, with major and minor ticks.

INPUT:

- lower_left - the lower left corner of the frame, as a list, tuple, or vector.
- upper_right - the upper right corner of the frame, as a list, tuple, or vector.
- ticks - (default: 4) the number of major ticks shown on each ruler.
- sub_ticks - (default: 4) the number of shown subdivisions between each major tick.

Type line3d.options for a dictionary of the default options for lines, which are also available.

EXAMPLES:

A ruler frame:

```
sage: from sage.plot.plot3d.shapes2 import ruler_frame
sage: F = ruler_frame([1,2,3],vector([2,3,4])); F
```

A ruler frame with some options:

```
sage: F = ruler_frame([1,2,3],vector([2,3,4]),ticks=6, sub_ticks=2, color='red'); F
```

sage.plot.plot3d.shapes2.**sphere**(*center=(0, 0, 0)*, *size=1*, ***kwds*)
Return a plot of a sphere of radius size centered at (x, y, z).

INPUT:

- `(x,y,z)` - center (default: (0,0,0)
- `size` - the radius (default: 1)

EXAMPLES: A simple sphere:

```
sage: sphere()
```

Two spheres touching:

```
sage: sphere(center=(-1,0,0)) + sphere(center=(1,0,0), aspect_ratio=[1,1,1])
```

Spheres of radii 1 and 2 one stuck into the other:

```
sage: sphere(color='orange') + sphere(color=(0,0,0.3), \
            center=(0,0,-2),size=2,opacity=0.9)
```

We draw a transparent sphere on a saddle.

```
sage: u,v = var('u v')
sage: saddle = plot3d(u^2 - v^2, (u,-2,2), (v,-2,2))
sage: sphere((0,0,1), color='red', opacity=0.5, aspect_ratio=[1,1,1]) + saddle
```

TESTS:

```
sage: T = sage.plot.plot3d.texture.Texture('red')
sage: S = sphere(texture=T)
sage: T in S.texture_set()
True
```

sage.plot.plot3d.shapes2.**text3d**(*txt*, *x_y_z*, ***kwds*)
Display 3d text.

INPUT:

- `txt` - some text
- `(x,y,z)` - position tuple (x, y, z)
- `**kwds` - standard 3d graphics options

Note: There is no way to change the font size or opacity yet.

EXAMPLES: We write the word Sage in red at position (1,2,3):

```
sage: text3d("Sage", (1,2,3), color=(0.5,0,0))
```

We draw a multicolor spiral of numbers:

```
sage: sum([text3d('%.1f'%n, (cos(n),sin(n),n), color=(n/2,1-n/2,0)) \
           for n in [0,0.2,..,8]])
```

Another example

```
sage: text3d("Sage is really neat!!",(2,12,1))
```

And in 3d in two places:

```
sage: text3d("Sage is...",(2,12,1), rgbcolor=(1,0,0)) + text3d("quite powerful!!",(4,10,0), rgbc
```

CHAPTER

ELEVEN

BASE CLASSES FOR 3D GRAPHICS OBJECTS AND PLOTTING

Base classes for 3D Graphics objects and plotting

AUTHORS:

- Robert Bradshaw (2007-02): initial version
- Robert Bradshaw (2007-08): Cythonization, much optimization
- William Stein (2008)

TODO: - finish integrating tachyon - good default lights, camera

class sage.plot.plot3d.base.**BoundingSphere**(*cen*, *r*)
Bases: sage.structure.sage_object.SageObject

A bounding sphere is like a bounding box, but is simpler to deal with and behaves better under rotations.

transform(*T*)
Returns the bounding sphere of this sphere acted on by T. This always returns a new sphere, even if the resulting object is an ellipsoid.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: from sage.plot.plot3d.base import BoundingSphere
sage: BoundingSphere((0,0,0), 10).transform(Transformation(trans=(1,2,3)))
Center (1.0, 2.0, 3.0) radius 10.0
sage: BoundingSphere((0,0,0), 10).transform(Transformation(scale=(1/2, 1, 2)))
Center (0.0, 0.0, 0.0) radius 20.0
sage: BoundingSphere((0,0,3), 10).transform(Transformation(scale=(2, 2, 2)))
Center (0.0, 0.0, 6.0) radius 20.0
```

class sage.plot.plot3d.base.**Graphics3d**
Bases: sage.structure.sage_object.SageObject

This is the baseclass for all 3d graphics objects.

aspect_ratio(*v=None*)
Sets or gets the preferred aspect ratio of self.

INPUT:

- v – (default: None) must be a list or tuple of length three, or the integer 1. If no arguments are provided then the default aspect ratio is returned.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.aspect_ratio([1,2,3])
sage: D.aspect_ratio()
[1.0, 2.0, 3.0]
sage: D.aspect_ratio(1)
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
```

bounding_box()

Returns the lower and upper corners of a 3d bounding box for self. This is used for rendering and self should fit entirely within this box.

Specifically, the first point returned should have x, y, and z coordinates should be the respective infimum over all points in self, and the second point is the supremum.

The default return value is simply the box containing the origin.

EXAMPLES:

```
sage: sphere((1,1,1), 2).bounding_box()
((-1.0, -1.0, -1.0), (3.0, 3.0, 3.0))
sage: G = line3d([(1, 2, 3), (-1,-2,-3)])
sage: G.bounding_box()
((-1.0, -2.0, -3.0), (1.0, 2.0, 3.0))
```

default_render_params()

Returns an instance of RenderParams suitable for plotting this object.

EXAMPLES:

```
sage: type(dodecahedron().default_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

export_jmol(*filename='jmol_shape.jmol', force_reload=False, zoom=100, spin=False, background=(1, 1, 1), stereo=False, mesh=False, dots=False, perspective_depth=True, orientation=(-764, -346, -545, 76.39), **ignored_kwds*)

A jmol scene consists of a script which refers to external files. Fortunately, we are able to put all of them in a single zip archive, which is the output of this call.

EXAMPLES:

```
sage: out_file = tmp_filename(ext=".jmol")
sage: G = sphere((1, 2, 3), 5) + cube() + sage.plot.plot3d.shapes.Text("hi")
sage: G.export_jmol(out_file)
sage: import zipfile
sage: z = zipfile.ZipFile(out_file)
sage: z.namelist()
['obj_...pmesh', 'SCRIPT']

sage: print z.read('SCRIPT')
data "model list"
2
empty
Xx 0 0 0
Xx 5.5 5.5 5.5
end "model list"; show data
select *
wireframe off; spacefill off
```

```
set labelOffset 0 0
background [255,255,255]
spin OFF
moveto 0 -764 -346 -545 76.39
centerAt absolute {0 0 0}
zoom 100
frank OFF
set perspectivedepth ON
isosurface sphere_1  center {1.0 2.0 3.0} sphere 5.0
color isosurface  [102,102,255]
pmesh obj_... "obj_...pmesh"
color pmesh  [102,102,255]
select atomno = 1
color atom  [102,102,255]
label "hi"
isosurface fullylit; pmesh o* fullylit; set antialiasdisplay on;

sage: print z.read(z.namelist()[0])
24
0.5 0.5 0.5
-0.5 0.5 0.5
...
-0.5 -0.5 -0.5
6
5
0
1
...
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

The generic Graphics3d object can't be made flatter.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.flatten() is G
True
```

frame_aspect_ratio(*v=None*)

Sets or gets the preferred frame aspect ratio of self.

INPUT:

- v – (default: None) must be a list or tuple of length three, or the integer 1. If no arguments are provided then the default frame aspect ratio is returned.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.frame_aspect_ratio([2,2,1])
sage: D.frame_aspect_ratio()
[2.0, 2.0, 1.0]
sage: D.frame_aspect_ratio(1)
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
```

jmol_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used by jmol to render self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may refer to several remove files, which are stored in render_parames.output_archive.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.jmol_repr(G.default_render_params())
[]
sage: G = sphere((1, 2, 3))
sage: G.jmol_repr(G.default_render_params())
[['isosurface sphere_1  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
```

json_repr(*render_params*)

A (possibly nested) list of strings. Each entry is formatted as JSON, so that a JavaScript client could eval it and get an object. Each object has fields to encapsulate the faces and vertices of self. This representation is intended to be consumed by the canvas3d viewer backend.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.json_repr(G.default_render_params())
[]
```

mtl_str()

Returns the contents of a .mtl file, to be used to provide coloring information for an .obj file.

EXAMPLES:: sage: G = tetrahedron(color='red') + tetrahedron(color='yellow', opacity=0.5) sage: print G.mtl_str() newmtl ... Ka 0.5 5e-06 5e-06 Kd 1.0 1e-05 1e-05 Ks 0.0 0.0 0.0 illum 1 Ns 1 d 1 newmtl ... Ka 0.5 0.5 5e-06 Kd 1.0 1.0 1e-05 Ks 0.0 0.0 0.0 illum 1 Ns 1 d 0.500000000000000

obj()

An .obj scene file (as a string) containing the this object. A .mtl file of the same name must also be produced for coloring.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import ColorCube
sage: print ColorCube(1, ['red', 'yellow', 'blue']).obj()
g obj_1
usemtl ...
v 1 1 1
v -1 1 1
v -1 -1 1
v 1 -1 1
f 1 2 3 4
...
g obj_6
usemtl ...
v -1 -1 1
v -1 1 1
v -1 1 -1
v -1 -1 -1
f 21 22 23 24
```

obj_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used to construct an .obj file of self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.obj_repr(G.default_render_params())
[]
sage: G = cube()
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0.5 0.5 0.5',
  'v -0.5 0.5 0.5',
  'v -0.5 -0.5 0.5',
  'v 0.5 -0.5 0.5',
  'v 0.5 0.5 -0.5',
  'v -0.5 0.5 -0.5',
  'v 0.5 -0.5 -0.5',
  'v -0.5 -0.5 -0.5'],
 ['f 1 2 3 4',
  'f 1 5 6 2',
  'f 1 4 7 5',
  'f 6 5 7 8',
  'f 7 4 3 8',
  'f 3 2 6 8'],
 []]
```

rotate(*v*, *theta*)

Returns self rotated about the vector v by $theta$ radians.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: v = (1,2,3)
sage: G = arrow3d((0, 0, 0), v)
sage: G += Cone(1/5, 1).translate((0, 0, 2))
sage: C = Cone(1/5, 1, opacity=.25).translate((0, 0, 2))
sage: G += sum(C.rotate(v, pi*t/4) for t in [1..7])
sage: G.show(aspect_ratio=1)

sage: from sage.plot.plot3d.shapes import Box
sage: Box(1/3, 1/5, 1/7).rotate((1, 1, 1), pi/3).show(aspect_ratio=1)
```

rotateX(*theta*)

Returns self rotated about the x-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateX(pi/2)
sage: G.show(aspect_ratio=1)
```

rotateY(*theta*)

Returns self rotated about the y-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateY(pi/3)
sage: G.show(aspect_ratio=1)
```

rotateZ(*theta*)

Returns self rotated about the z-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
sage: G = Box(1/2, 1/3, 1/5) + Box(1/2, 1/3, 1/5, opacity=.25).rotateZ(pi/5)
sage: G.show(aspect_ratio=1)
```

save (*filename*, ***kwds*)

Save the graphic to an image file (of type: PNG, BMP, GIF, PPM, or TIFF) rendered using Tachyon, or pickle it (stored as an SOBJ so you can load it later) depending on the file extension you give the filename.

INPUT:

- `filename` - Specify where to save the image or object.
- `**kwds` - When specifying an image file to be rendered by Tachyon, any of the viewing options accepted by show() are valid as keyword arguments to this function and they will behave in the same way. Accepted keywords include: `viewer`, `verbosity`, `figsize`, `aspect_ratio`, `frame_aspect_ratio`, `zoom`, `frame`, and `axes`. Default values are provided.

EXAMPLES:

```
sage: f = tmp_filename() + '.png'
sage: G = sphere()
sage: G.save(f)
```

We demonstrate using keyword arguments to control the appearance of the output image:

```
sage: G.save(f, zoom=2, figsize=[5, 10])
```

But some extra parameters don't make since (like `viewer`, since rendering is done using Tachyon only). They will be ignored:

```
sage: G.save(f, viewer='jmol') # Looks the same
```

Since Tachyon only outputs PNG images, PIL will be used to convert to alternate formats:

```
sage: cube().save(tmp_filename(ext='.gif'))
```

save_image (*filename=None*, **args*, ***kwds*)

Save an image representation of self. The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`. Currently this is implemented by calling the `save()` method of self, passing along all arguments and keywords.

Note: Not all image types are necessarily implemented for all graphics types. See `save()` for more details.

EXAMPLES:

```
sage: f = tmp_filename() + '.png'
sage: G = sphere()
sage: G.save_image(f)
```

scale (**x*)

Returns self scaled in the x, y, and z directions.

EXAMPLES:

```
sage: G = dodecahedron() + dodecahedron(opacity=.5).scale(2)
sage: G.show(aspect_ratio=1)
sage: G = icosahedron() + icosahedron(opacity=.5).scale([1, 1/2, 2])
sage: G.show(aspect_ratio=1)
```

TESTS:

```
sage: G = sphere((0, 0, 0), 1)
sage: G.scale(2)
sage: G.scale(1, 2, 1/2).show(aspect_ratio=1)
sage: G.scale(2).bounding_box()
((-2.0, -2.0, -2.0), (2.0, 2.0, 2.0))
```

show(***kwds*)

INPUT:

- `viewer` - string (default: 'jmol'), how to view the plot
  - 'jmol': Interactive 3D viewer using Java
  - 'tachyon': Ray tracer generates a static PNG image
  - 'java3d': Interactive OpenGL based 3D
  - 'canvas3d': Web-based 3D viewer powered by JavaScript and <canvas> (notebook only)
- `filename` - string (default: a temp file); file to save the image to
- `verbosity` - display information about rendering the figure
- `figsize` - (default: 5); x or pair [x,y] for numbers, e.g., [5,5]; controls the size of the output figure. E.g., with Tachyon the number of pixels in each direction is 100 times figsize[0]. This is ignored for the jmol embedded renderer.
- `aspect_ratio` - (default: "automatic") - aspect ratio of the coordinate system itself. Give [1,1,1] to make spheres look round.
- `frame_aspect_ratio` - (default: "automatic") aspect ratio of frame that contains the 3d scene.
- `zoom` - (default: 1) how zoomed in
- `frame` - (default: True) if True, draw a bounding frame with labels
- `axes` - (default: False) if True, draw coordinate axes
- `**kwds` - other options, which make sense for particular rendering engines

CHANGING DEFAULTS: Defaults can be uniformly changed by importing a dictionary and changing it. For example, here we change the default so images display without a frame instead of with one:

```
sage: from sage.plot.plot3d.base import SHOW_DEFAULTS
sage: SHOW_DEFAULTS['frame'] = False
```

This sphere will not have a frame around it:

```
sage: sphere((0,0,0))
```

We change the default back:

```
sage: SHOW_DEFAULTS['frame'] = True
```

Now this sphere is enclosed in a frame:

```
sage: sphere((0,0,0))
```

EXAMPLES: We illustrate use of the `aspect_ratio` option:

```
sage: x, y = var('x,y')
sage: p = plot3d(2*sin(x*y), (x, -pi, pi), (y, -pi, pi))
sage: p.show(aspect_ratio=[1,1,1])
```

This looks flattened, but filled with the plot:

```
sage: p.show(frame_aspect_ratio=[1,1,1/16])
```

This looks flattened, but the plot is square and smaller:

```
sage: p.show(aspect_ratio=[1,1,1], frame_aspect_ratio=[1,1,1/8])
```

This example shows indirectly that the defaults from `plot()` are dealt with properly:

```
sage: plot(vector([1,2,3]))
```

We use the 'canvas3d' backend from inside the notebook to get a view of the plot rendered inline using HTML canvas:

```
sage: p.show(viewer='canvas3d')
```

tachyon()

An tachyon input file (as a string) containing the this object.

EXAMPLES:

```
sage: print sphere((1, 2, 3), 5, color='yellow').tachyon()
begin_scene
resolution 400 400
         camera
        ...
      plane
        center -2000 -1000 -500
        normal 2.3 2.4 2.0
        TEXTURE
            AMBIENT 1.0 DIFFUSE 1.0 SPECULAR 1.0 OPACITY 1.0
            COLOR 1.0 1.0 1.0
            TEXFUNC 0
    Texdef texture...
  Ambient 0.333333333333 Diffuse 0.666666666667 Specular 0.0 Opacity 1
   Color 1.0 1.0 0.0
   TexFunc 0
    Sphere center 1.0 -2.0 3.0 Rad 5.0 texture...
end_scene

sage: G = icosahedron(color='red') + sphere((1,2,3), 0.5, color='yellow')
sage: G.show(viewer='tachyon', frame=false)
sage: print G.tachyon()
begin_scene
...
Texdef texture...
  Ambient 0.333333333333 Diffuse 0.666666666667 Specular 0.0 Opacity 1
   Color 1.0 1.0 0.0
   TexFunc 0
TRI V0 ...
Sphere center 1.0 -2.0 3.0 Rad 0.5 texture...
end_scene
```

tachyon_repr(*render_params*)

A (possibly nested) list of strings which will be concatenated and used by tachyon to render self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.tachyon_repr(G.default_render_params())
[]
sage: G = sphere((1, 2, 3))
sage: G.tachyon_repr(G.default_render_params())
['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']
```

testing_render_params()

Returns an instance of RenderParams suitable for testing this object. In particular, it opens up '/dev/null' as an auxiliary zip file for jmol.

EXAMPLES:

```
sage: type(dodecahedron().testing_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

texture

texture_set()

Often the textures of a 3d file format are kept separate from the objects themselves. This function returns the set of textures used, so they can be defined in a preamble or separate file.

EXAMPLES:

```
sage: sage.plot.plot3d.base.Graphics3d().texture_set()
set([])

sage: G = tetrahedron(color='red') + tetrahedron(color='yellow') + tetrahedron(color='red',
sage: [t for t in G.texture_set() if t.color == colors.red] # we should have two red texture
[Texture(texture..., red, ff0000), Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # ...and one yellow
[Texture(texture..., yellow, ffff00)]
```

transform(***kwds*)

Apply a transformation to self, where the inputs are passed onto a TransformGroup object. Mostly for internal use; see the translate, scale, and rotate methods for more details.

EXAMPLES:

```
sage: sphere((0,0,0), 1).transform(trans=(1, 0, 0), scale=(2,3,4)).bounding_box()
((-1.0, -3.0, -4.0), (3.0, 3.0, 4.0))
```

translate(**x*)

Return self translated by the given vector (which can be given either as a 3-iterable or via positional arguments).

EXAMPLES:

```
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate(2*n, 0, 0) for n in [1..4])
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate([-2*n, n, n^2]) for n in [1..4
```

TESTS:

```
sage: G = sphere((0, 0, 0), 1)
sage: G.bounding_box()
((-1.0, -1.0, -1.0), (1.0, 1.0, 1.0))
sage: G.translate(0, 0, 1).bounding_box()
((-1.0, -1.0, 0.0), (1.0, 1.0, 2.0))
sage: G.translate(-1, 5, 0).bounding_box()
((-2.0, 4.0, -1.0), (0.0, 6.0, 1.0))
```

viewpoint()
Returns the viewpoint of this plot. Currently only a stub for x3d.

EXAMPLES:

```
sage: type(dodecahedron().viewpoint())
<class 'sage.plot.plot3d.base.Viewpoint'>
```

x3d()
An x3d scene file (as a string) containing the this object.

EXAMPLES:

```
sage: print sphere((1, 2, 3), 5).x3d()
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Transform translation='1 2 3'>
<Shape><Sphere radius='5.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
</Scene>
</X3D>

sage: G = icosahedron() + sphere((0,0,0), 0.5, color='red')
sage: print G.x3d()
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Shape>
<IndexedFaceSet coordIndex='...'>
  <Coordinate point='...'/>
</IndexedFaceSet>
<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1' specularColor='0.0 0.0 0.0'/>
<Transform translation='0 0 0'>
<Shape><Sphere radius='0.5'/><Appearance><Material diffuseColor='1.0 0.0 0.0' shininess='1'
</Transform>
</Scene>
</X3D>
```

class sage.plot.plot3d.base.**Graphics3dGroup**(*all=()*, *rot=None*, *trans=None*, *scale=None*, *T=None*)
Bases: sage.plot.plot3d.base.Graphics3d

This class represents a collection of 3d objects. Usually they are formed implicitly by summing.

bounding_box()
Box that contains the bounding boxes of all the objects that make up self.

EXAMPLES:

```
sage: A = sphere((0,0,0), 5)
sage: B = sphere((1, 5, 10), 1)
sage: A.bounding_box()
((-5.0, -5.0, -5.0), (5.0, 5.0, 5.0))
sage: B.bounding_box()
((0.0, 4.0, 9.0), (2.0, 6.0, 11.0))
```

```
sage: (A+B).bounding_box()
((-5.0, -5.0, -5.0), (5.0, 6.0, 11.0))
sage: (A+B).show(aspect_ratio=1, frame=True)

sage: sage.plot.plot3d.base.Graphics3dGroup([]).bounding_box()
((0.0, 0.0, 0.0), (0.0, 0.0, 0.0))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sum([circle((0, 0), t) for t in [1..10]], sphere()); G
sage: G.flatten()
sage: len(G.all)
2
sage: len(G.flatten().all)
11
```

jmol_repr(*render_params*)

The jmol representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.jmol_repr(G.default_render_params())
[['isosurface sphere_1  center {0.0 0.0 0.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
 [['isosurface sphere_2  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
```

json_repr(*render_params*)

The JSON representation of a group is simply the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1, 2, 3))
sage: G.json_repr(G.default_render_params())
[["{vertices:..."]], [["{vertices:..."]]]
```

obj_repr(*render_params*)

The obj representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = tetrahedron() + tetrahedron().translate(10, 10, 10)
sage: G.obj_repr(G.default_render_params())
[['g obj_1',
  'usemtl ...',
  ['v 0 0 1',
   'v 0.942809 0 -0.333333',
   'v -0.471405 0.816497 -0.333333',
   'v -0.471405 -0.816497 -0.333333'],
  ['f 1 2 3', 'f 2 4 3', 'f 1 3 4', 'f 1 4 2'],
  []],
 [['g obj_2',
   'usemtl ...',
   ['v 10 10 11',
    'v 10.9428 10 9.66667',
    'v 9.5286 10.8165 9.66667',
    'v 9.5286 9.1835 9.66667'],
   ['f 5 6 7', 'f 6 8 7', 'f 5 7 8', 'f 5 8 6'],
   []]]]
```

set_texture(***kwds*)

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron((3, 0, 0), color='blue')
sage: G
sage: G.set_texture(color='yellow')
sage: G
```

tachyon_repr(*render_params*)

The tachyon representation of a group is simply the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 0.0 0.0 0.0 Rad 1.0 texture...'],
 ['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']]
```

texture_set()

The texture set of a group is simply the union of the textures of all its objects.

EXAMPLES:

```
sage: G = sphere(color='red') + sphere(color='yellow')
sage: [t for t in G.texture_set() if t.color == colors.red] # one red texture
[Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # one yellow texture
[Texture(texture..., yellow, ffff00)]

sage: T = sage.plot.plot3d.texture.Texture('blue'); T
Texture(texture..., blue, 0000ff)
sage: G = sphere(texture=T) + sphere((1, 1, 1), texture=T)
sage: len(G.texture_set())
1
```

transform(***kwds*)

Transforming this entire group simply makes a transform group with the same contents.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
sage: G.transform(scale=(2,1/2,1))
sage: G.transform(trans=(1,1,3))
```

x3d_str()

The x3d representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: print G.x3d_str()
<Transform translation='0 0 0'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
<Transform translation='1 2 3'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
```

class sage.plot.plot3d.base.**PrimitiveObject**

Bases: sage.plot.plot3d.base.Graphics3d

This is the base class for the non-container 3d objects.

get_texture()

EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.get_texture()
Texture(texture..., red, ff0000)
```

jmol_repr(*render_params*)

Default behavior is to render the triangulation. The actual polygon data is stored in a separate file.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.jmol_repr(G.testing_render_params())
['pmesh obj_1 "obj_1.pmesh"\ncolor pmesh  [102,102,255]']
```

obj_repr(*render_params*)

Default behavior is to render the triangulation.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0 1 0.5',
 ...
  'f ...'],
 []]
```

set_texture(*texture=None*, ***kwds*)

EXAMPLES:

```
sage: G = dodecahedron(color='red'); G
sage: G.set_texture(color='yellow'); G
```

tachyon_repr(*render_params*)

Default behavior is to render the triangulation.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.tachyon_repr(G.default_render_params())
['TRI V0 0 1 0.5
...
'texture...']
```

texture_set()

EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.texture_set()
set([Texture(texture..., red, ff0000)])
```

x3d_str()

EXAMPLES:

```
sage: sphere().flatten().x3d_str()
"<Transform>\n<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0'
```

class sage.plot.plot3d.base.**RenderParams**(***kwds*)

Bases: sage.structure.sage_object.SageObject

This class is a container for all parameters that may be needed to render triangulate/render an object to a certain format. It can contain both cumulative and global parameters.

Of particular note is the transformation object, which holds the cumulative transformation from the root of the scene graph to this node in the tree.

pop_transform()

Remove the last transformation off the stack, resetting self.transform to the previous value.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: T = Transformation(trans=(100, 500, 0))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
sage: params.push_transform(Transformation(trans=(-100, 500, 200)))
sage: params.transform.get_matrix()
[   1.0    0.0    0.0    0.0]
[   0.0    1.0    0.0 1000.0]
[   0.0    0.0    1.0  200.0]
[   0.0    0.0    0.0    1.0]
sage: params.pop_transform()
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
```

push_transform(*T*)

Push a transformation onto the stack, updating self.transform.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.transform is None
True
sage: T = Transformation(scale=(10,20,30))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[10.0  0.0  0.0  0.0]
[ 0.0 20.0  0.0  0.0]
[ 0.0  0.0 30.0  0.0]
[ 0.0  0.0  0.0  1.0]
sage: params.push_transform(T)  # scale again
sage: params.transform.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 400.0   0.0   0.0]
[  0.0   0.0 900.0   0.0]
```

```
[  0.0   0.0   0.0   1.0]
```

unique_name(*desc='name'*)

Returns a unique identifier starting with desc.

EXAMPLES:

```
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.unique_name()
'name_1'
sage: params.unique_name()
'name_2'
sage: params.unique_name('texture')
'texture_3'
```

class sage.plot.plot3d.base.**TransformGroup**(*all=[]*, *rot=None*, *trans=None*, *scale=None*, *T=None*)

Bases: sage.plot.plot3d.base.Graphics3dGroup

This class is a container for a group of objects with a common transformation.

bounding_box()

Returns the bounding box of self, i.e. the box containing the contents of self after applying the transformation.

EXAMPLES:

```
sage: G = cube()
sage: G.bounding_box()
((-0.5, -0.5, -0.5), (0.5, 0.5, 0.5))
sage: G.scale(4).bounding_box()
((-2.0, -2.0, -2.0), (2.0, 2.0, 2.0))
sage: G.rotateZ(pi/4).bounding_box()
((-0.7071067811865475, -0.7071067811865475, -0.5),
 (0.7071067811865475, 0.7071067811865475, 0.5))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]

sage: G.flatten().get_transformation().get_matrix()
[100.0   0.0   0.0 100.0]
[  0.0 100.0   0.0 200.0]
[  0.0   0.0 100.0 300.0]
[  0.0   0.0   0.0   1.0]
```

get_transformation()

Returns the actual transformation object associated with self.

EXAMPLES:

```
sage: G = sphere().scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]
```

jmol_repr(*render_params*)

Transformations for jmol are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.jmol_repr(G.default_render_params())
[[['isosurface sphere_1  center {2.0 4.0 6.0} sphere 2.0\ncolor isosurface  [102,102,255]']]
```

json_repr(*render_params*)

Transformations are applied at the leaf nodes.

EXAMPLES:

```
sage: G = cube().rotateX(0.2)
sage: G.json_repr(G.default_render_params())
[["{vertices:[{x:0.5,y:0.589368,z:0.390699},..."]]
```

obj_repr(*render_params*)

Transformations for .obj files are applied at the leaf nodes.

EXAMPLES:

```
sage: G = cube().scale(4).translate(1, 2, 3)
sage: G.obj_repr(G.default_render_params())
[['g obj_1',
  'usemtl ...',
  ['v 3 4 5',
   'v -1 4 5',
   'v -1 0 5',
   'v 3 0 5',
   'v 3 4 1',
   'v -1 4 1',
   'v 3 0 1',
   'v -1 0 1'],
  ['f 1 2 3 4',
   'f 1 5 6 2',
   'f 1 4 7 5',
   'f 6 5 7 8',
   'f 7 4 3 8',
   'f 3 2 6 8'],
  []]]
```

tachyon_repr(*render_params*)

Transformations for Tachyon are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 2.0 4.0 6.0 Rad 2.0 texture...']]
```

transform(***kwds*)

Transforming this entire group can be done by composing transformations.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
sage: G.transform(scale=(2,1/2,1))
sage: G.transform(trans=(1,1,3))
```

x3d_str()

To apply a transformation to a set of objects in x3d, simply make them all children of an x3d Transform node.

EXAMPLES:

```
sage: sphere((1,2,3)).x3d_str()
"<Transform translation='1 2 3'>\n<Shape><Sphere radius='1.0'/><Appearance><Material diffuse
```

class sage.plot.plot3d.base.**Viewpoint**(**x*)

Bases: sage.plot.plot3d.base.Graphics3d

This class represents a viewpoint, necessary for x3d.

In the future, there could be multiple viewpoints, and they could have more properties. (Currently they only hold a position).

x3d_str()

EXAMPLES:

```
sage: sphere((0,0,0), 100).viewpoint().x3d_str()
"<Viewpoint position='0 0 6'/>"
```

sage.plot.plot3d.base.**flatten_list**(*L*)

This is an optimized routine to turn a list of lists (of lists ...) into a single list. We generate data in a non-flat format to avoid multiple data copying, and then concatenate it all at the end.

This is NOT recursive, otherwise there would be a lot of redundant copying (which we are trying to avoid in the first place, though at least it would be just the pointers).

EXAMPLES:

```
sage: from sage.plot.plot3d.base import flatten_list
sage: flatten_list([])
[]
sage: flatten_list([[[[]]]])
[]
sage: flatten_list([['a', 'b'], 'c'])
['a', 'b', 'c']
sage: flatten_list([['a'], [['b'], 'c'], ['d'], [['e', 'f', 'g']]])
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

sage.plot.plot3d.base.**max3**(*v*)

Return the componentwise maximum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: max3([(-1,2,5), (-3, 4, 2)])
(-1, 4, 5)
```

sage.plot.plot3d.base.**min3**(*v*)
Return the componentwise minimum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: min3([(-1,2,5), (-3, 4, 2)])
(-3, 2, 2)
```

sage.plot.plot3d.base.**optimal_aspect_ratios**(*ratios*)

sage.plot.plot3d.base.**optimal_extra_kwds**(*v*)
Given a list v of dictionaries, this function merges them such that later dictionaries have precedence.

sage.plot.plot3d.base.**point_list_bounding_box**(*v*)
EXAMPLES:

```
sage: from sage.plot.plot3d.base import point_list_bounding_box
sage: point_list_bounding_box([(1,2,3),(4,5,6),(-10,0,10)])
((-10.0, 0.0, 3.0), (4.0, 5.0, 10.0))
sage: point_list_bounding_box([(float('nan'), float('inf'), float('-inf')), (10,0,10)])
((10.0, 0.0, 10.0), (10.0, 0.0, 10.0))
```

CHAPTER

TWELVE

THE TACHYON 3D RAY TRACER

Given any 3D graphics object one can compute a raytraced representation by typing `show(viewer='tachyon')`. For example, we draw two translucent spheres that contain a red tube, and render the result using Tachyon.

```
sage: S = sphere(opacity=0.8, aspect_ratio=[1,1,1])
sage: L = line3d([(0,0,0),(2,0,0)], thickness=10, color='red')
sage: M = S + S.translate((2,0,0)) + L
sage: M.show(viewer='tachyon')
```

One can also directly control Tachyon, which gives a huge amount of flexibility. For example, here we directly use Tachyon to draw 3 spheres on the coordinate axes. Notice that the result is gorgeous:

```
sage: t = Tachyon(xres=500,yres=500, camera_center=(2,0,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t3', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t4', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: t.sphere((0,0.5,0), 0.2, 't2')
sage: t.sphere((0.5,0,0), 0.2, 't3')
sage: t.sphere((0,0,0.5), 0.2, 't4')
sage: t.show()
```

AUTHOR:

- John E. Stone (johns@megapixel.com): wrote tachyon ray tracer
- William Stein: sage-tachyon interface
- Joshua Kantor: 3d function plotting
- Tom Boothby: 3d function plotting n'stuff
- Leif Hille: key idea for bugfix for texfunc issue (trac #799)
- Marshall Hampton: improved doctests, rings, axis-aligned boxes.

TODO:

- clean up trianglefactory stuff

class `sage.plot.plot3d.tachyon.`**`Axis_aligned_box`**(*min_p*, *max_p*, *texture*)

Box with axis-aligned edges with the given min and max coordinates.

`str`()

Returns the scene string of the axis-aligned box.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Axis_aligned_box
sage: aab = Axis_aligned_box((0,0,0),(1,1,1),'s')
sage: aab.str()
'\n        box min  0.0 0.0 0.0  max  1.0 1.0 1.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Cylinder**(*center*, *axis*, *radius*, *texture*)
An infinite cylinder.

str()
Returns the scene string of the cylinder.

EXAMPLES:

```
sage: t = Tachyon()
sage: from sage.plot.plot3d.tachyon import Cylinder
sage: c = Cylinder((0,0,0),(1,1,1),.1,'s')
sage: c.str()
'\n        cylinder center  0.0 0.0 0.0  axis  1.0 1.0 1.0  rad 0.1 s\n        '
```

class sage.plot.plot3d.tachyon.**FCylinder**(*base*, *apex*, *radius*, *texture*)
A finite cylinder.

str()
Returns the scene string of the finite cylinder.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import FCylinder
sage: fc = FCylinder((0,0,0),(1,1,1),.1,'s')
sage: fc.str()
'\n        fcylinder base  0.0 0.0 0.0  apex  1.0 1.0 1.0  rad 0.1 s\n        '
```

class sage.plot.plot3d.tachyon.**FractalLandscape**(*res*, *scale*, *center*, *texture*)
Axis-aligned fractal landscape. Does not seem very useful at the moment, but perhaps will be improved in the future.

str()
Returns the scene string of the fractal landscape.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import FractalLandscape
sage: fl = FractalLandscape([20,20],[30,30],[1,2,3],'s')
sage: fl.str()
'\n        scape res  20 20  scale  30 30  center  1.0 2.0 3.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Light**(*center*, *radius*, *color*)
Represents lighting objects.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Light
sage: q = Light((1,1,1),1,(1,1,1))
sage: q._center
(1, 1, 1)
```

str()
Returns the tachyon string defining the light source.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Light
sage: q = Light((1,1,1),1,(1,1,1))
sage: q._radius
1
```

class sage.plot.plot3d.tachyon.**ParametricPlot**(*f*, *t_0*, *t_f*, *tex*, *r=0.1*, *cylinders=True*, *min_depth=4*, *max_depth=8*, *e_rel=0.01*, *e_abs=0.01*)

Parametric plotting routines.

str()

Returns the tachyon string representation of the parameterized curve.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import ParametricPlot
sage: t = var('t')
sage: f = lambda t: (t,t^2,t^3)
sage: q = ParametricPlot(f,0,1,'s')
sage: q.str()[9:69]
'sphere center  0.0 0.0 0.0  rad 0.1 s\n        \n        fcyli'
```

tol(*est*, *val*)

Check relative, then absolute tolerance. If both fail, return False. This is a zero-safe error checker.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import ParametricPlot
sage: t = var('t')
sage: f = lambda t: (t,t^2,t^3)
sage: q = ParametricPlot(f,0,1,'s')
sage: q.tol([0,0,0],[1,0,0])
False
sage: q.tol([0,0,0],[.0001,0,0])
True
```

class sage.plot.plot3d.tachyon.**Plane**(*center*, *normal*, *texture*)

An infinite plane.

str()

Returns the scene string of the plane.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Plane
sage: p = Plane((1,2,3),(1,2,4),'s')
sage: p.str()
'\n        plane center  1.0 2.0 3.0  normal  1.0 2.0 4.0  s\n        '
```

class sage.plot.plot3d.tachyon.**Ring**(*center*, *normal*, *inner*, *outer*, *texture*)

An annulus of zero thickness.

str()

Returns the scene string of the ring.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Ring
sage: r = Ring((0,0,0), (1,1,0), 1.0, 2.0, 's')
sage: r.str()
'\n        ring center  0.0 0.0 0.0  normal  1.0 1.0 0.0  inner 1.0 outer 2.0 s\n        '
```

class sage.plot.plot3d.tachyon.**Sphere**(*center*, *radius*, *texture*)

A class for creating spheres in tachyon.

str()

Returns the scene string for the sphere.

EXAMPLES:

```
sage: t = Tachyon()
sage: from sage.plot.plot3d.tachyon import Sphere
sage: t.texture('r', color=(.8,0,0), ambient = .1)
sage: s = Sphere((1,1,1),1,'r')
sage: s.str()
'\n        sphere center  1.0 1.0 1.0  rad 1.0 r\n        '
```

class sage.plot.plot3d.tachyon.**Tachyon**(*xres=350*, *yres=350*, *zoom=1.0*, *antialiasing=False*, *aspectratio=1.0*, *raydepth=8*, *camera_center=(-3, 0, 0)*, *updir=(0, 0, 1)*, *look_at=(0, 0, 0)*, *viewdir=None*, *projection='PERSPECTIVE'*)

Bases: sage.structure.sage_object.SageObject

Create a scene the can be rendered using the Tachyon ray tracer.

INPUT:

- xres - (default 350)
- yres - (default 350)
- zoom - (default 1.0)
- antialiasing - (default False)
- aspectratio - (default 1.0)
- raydepth - (default 5)
- camera_center - (default (-3, 0, 0))
- updir - (default (0, 0, 1))
- look_at - (default (0,0,0))
- viewdir - (default None)
- projection - (default 'PERSPECTIVE')

OUTPUT: A Tachyon 3d scene.

Note that the coordinates are by default such that z is up, positive y is to the {left} and x is toward you. This is not oriented according to the right hand rule.

EXAMPLES: Spheres along the twisted cubic.

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(3,0.3,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2,diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: k=0
sage: for i in srange(-1,1,0.05):
....:     k += 1
....:     t.sphere((i,i^2-0.5,i^3), 0.1, 't%s'%(k%3))
sage: t.show()
```

Another twisted cubic, but with a white background, got by putting infinite planes around the scene.

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(3,0.3,0), raydepth=8)
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2,diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: t.texture('white', color=(1,1,1))
sage: t.plane((0,0,-1), (0,0,1), 'white')
sage: t.plane((0,-20,0), (0,1,0), 'white')
sage: t.plane((-20,0,0), (1,0,0), 'white')

sage: k=0
sage: for i in srange(-1,1,0.05):
....:     k += 1
....:     t.sphere((i,i^2 - 0.5,i^3), 0.1, 't%s'%(k%3))
....:     t.cylinder((0,0,0), (0,0,1), 0.05,'t1')
sage: t.show()
```

Many random spheres:

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(2,0.5,0.5), look_at=(0.5,0.5,0.5), raydepth=
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2, diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: k=0
sage: for i in range(100):
....:     k += 1
....:     t.sphere((random(),random(), random()), random()/10, 't%s'%(k%3))
sage: t.show()
```

Points on an elliptic curve, their height indicated by their height above the axis:

```
sage: t = Tachyon(camera_center=(5,2,2), look_at=(0,1,0))
sage: t.light((10,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: E = EllipticCurve('37a')
sage: P = E([0,0])
sage: Q = P
sage: n = 100
sage: for i in range(n):   # increase 20 for a better plot
....:     Q = Q + P
....:     t.sphere((Q[1], Q[0], ZZ(i)/n), 0.1, 't%s'%(i%3))
sage: t.show()
```

A beautiful picture of rational points on a rank 1 elliptic curve.

```
sage: t = Tachyon(xres=1000, yres=800, camera_center=(2,7,4), look_at=(2,0,0), raydepth=4)
sage: t.light((10,3,2), 1, (1,1,1))
sage: t.light((10,-3,2), 1, (1,1,1))
sage: t.texture('black', color=(0,0,0))
sage: t.texture('red', color=(1,0,0))
sage: t.texture('grey', color=(.9,.9,.9))
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.cylinder((0,0,0),(1,0,0),.01,'black')
sage: t.cylinder((0,0,0),(0,1,0),.01,'black')
sage: E = EllipticCurve('37a')
```

```
sage: P = E([0,0])
sage: Q = P
sage: n = 100
sage: for i in range(n):
....:     Q = Q + P
....:     c = i/n + .1
....:     t.texture('r%s'%i,color=(float(i/n),0,0))
....:     t.sphere((Q[0], -Q[1], .01), .04, 'r%s'%i)
sage: t.show()     # long time, e.g., 10-20 seconds
```

A beautiful spiral.

```
sage: t = Tachyon(xres=800,yres=800, camera_center=(2,5,2), look_at=(2.5,0,0))
sage: t.light((0,0,100), 1, (1,1,1))
sage: t.texture('r', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: for i in srange(0,50,0.1):
....:     t.sphere((i/10,sin(i),cos(i)), 0.05, 'r')
sage: t.texture('white', color=(1,1,1), opacity=1, specular=1, diffuse=1)
sage: t.plane((0,0,-100), (0,0,-100), 'white')
sage: t.show()
```

axis_aligned_box(*min_p*, *max_p*, *texture*)

Creates an axis-aligned box with minimal point `min_p` and maximum point `max_p`.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.axis_aligned_box((0,0,0),(2,2,2),'s')
```

cylinder(*center*, *axis*, *radius*, *texture*)

Creates the scene information for a infinite cylinder with the given center, axis direction, radius, and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('c')
sage: t.cylinder((0,0,0),(-1,-1,-1),.1,'c')
```

fcylinder(*base*, *apex*, *radius*, *texture*)

Finite cylinders are almost the same as infinite ones, but the center and length of the axis determine the extents of the cylinder. The finite cylinder is also really a shell, it doesn't have any caps. If you need to close off the ends of the cylinder, use two ring objects, with the inner radius set to 0.0 and the normal set to be the axis of the cylinder. Finite cylinders are built this way to enhance speed.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.fcylinder((1,1,1),(1,2,3),.01,'s')
sage: len(t.str())
423
```

fractal_landscape(*res*, *scale*, *center*, *texture*)

Axis-aligned fractal landscape. Not very useful at the moment.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: t.fractal_landscape([30,30],[80,80],[0,0,0],'s')
```

```
sage: len(t._objects)
2
```

light (*center*, *radius*, *color*)

Creates a light source of the given center, radius, and color.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,1),1.0,(.2,0,.8))
sage: q.str().split('\n')[17]
'        light center  1.0 1.0 1.0 '
```

parametric_plot (*f*, *t_0*, *t_f*, *tex*, *r=0.1*, *cylinders=True*, *min_depth=4*, *max_depth=8*, *e_rel=0.01*, *e_abs=0.01*)

Plots a space curve as a series of spheres and finite cylinders. Example (twisted cubic)

```
sage: f = lambda t: (t,t^2,t^3)
sage: t = Tachyon(camera_center=(5,0,4))
sage: t.texture('t')
sage: t.light((-20,-20,40), 0.2, (1,1,1))
sage: t.parametric_plot(f,-5,5,'t',min_depth=6)
sage: t.show(verbose=1)
tachyon ...
Scene contains 514 objects.
...
```

plane (*center*, *normal*, *texture*)

Creates an infinite plane with the given center and normal.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.plane((0,0,0),(1,1,1),'s')
sage: t.str()[338:380]
'plane center  0.0 0.0 0.0  normal  1.0 1.0'
```

plot (*f*, *xmin_xmax*, *ymin_ymax*, *texture*, *grad_f=None*, *max_bend=0.7*, *max_depth=5*, *initial_depth=3*, *num_colors=None*)

INPUT:

- `f` - Function of two variables, which returns a float (or coercible to a float) (xmin,xmax)
- `(ymin,ymax)` - defines the rectangle to plot over texture: Name of texture to be used Optional arguments:
- `grad_f` - gradient function. If specified, smooth triangles will be used.
- `max_bend` - Cosine of the threshold angle between triangles used to determine whether or not to recurse after the minimum depth
- `max_depth` - maximum recursion depth. Maximum triangles plotted = 2^{2*max_depth}
- `initial_depth` - minimum recursion depth. No error-tolerance checking is performed below this depth. Minimum triangles plotted: 2^{2*min_depth}
- `num_colors` - Number of rainbow bands to color the plot with. Texture supplied will be cloned (with different colors) using the texture_recolor method of the Tachyon object.

Plots a function by constructing a mesh with nonstandard sampling density without gaps. At very high resolutions (depths 10) it becomes very slow. Cython may help. Complexity is approx. $O(2^{2*maxdepth})$.

This algorithm has been optimized for speed, not memory - values from f(x,y) are recycled rather than calling the function multiple times. At high recursion depth, this may cause problems for some machines.

Flat Triangles:

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(4,-4,3),viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,  opacity=1.0, color=(1.0,0,0))
sage: t.plot(f,(-4,4),(-4,4),"t0",max_depth=5,initial_depth=3, num_colors=60)  # increase mi
sage: t.show(verbose=1)
tachyon ...
Scene contains 2713 objects.
...
```

Plotting with Smooth Triangles (requires explicit gradient function):

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(4,-4,3),viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: def g(x,y): return ( float(y*cos(x*y)), float(x*cos(x*y)), 1 )
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,  opacity=1.0, color=(1.0,0,0))
sage: t.plot(f,(-4,4),(-4,4),"t0",max_depth=5,initial_depth=3, grad_f = g)  # increase min_d
sage: t.show(verbose=1)
tachyon ...
Scene contains 2713 objects.
...
```

Preconditions: f is a scalar function of two variables, grad_f is None or a triple-valued function of two variables, min_x != max_x, min_y != max_y

```
sage: f = lambda x,y: x*y
sage: t = Tachyon()
sage: t.plot(f,(2.,2.),(-2.,2.),'')
Traceback (most recent call last):
...
ValueError: Plot rectangle is really a line.  Make sure min_x != max_x and min_y != max_y.
```

ring (*center*, *normal*, *inner*, *outer*, *texture*)

Creates the scene information for a ring with the given parameters.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.ring([0,0,0], [0,0,1], 1.0, 2.0, 's')
sage: t._objects[0]._center
[0, 0, 0]
```

save (*filename='sage.png'*, *verbose=0*, *block=True*, *extra_opts=''*)

INPUT:

- `filename` - (default: 'sage.png') output filename; the extension of the filename determines the type. Supported types include:
- `tga` - 24-bit (uncompressed)
- `bmp` - 24-bit Windows BMP (uncompressed)
- `ppm` - 24-bit PPM (uncompressed)
- `rgb` - 24-bit SGI RGB (uncompressed)
- `png` - 24-bit PNG (compressed, lossless)

- `verbose` - integer; (default: 0)
- `0` - silent
- `1` - some output
- `2` - very verbose output
- `block` - bool (default: True); if False, run the rendering command in the background.
- `extra_opts` - passed directly to tachyon command line. Use tachyon_rt.usage() to see some of the possibilities.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,11), 1,(1,1,1))
sage: q.texture('s')
sage: q.sphere((0,0,0),1,'s')
sage: tempname = tmp_filename()
sage: q.save(tempname)
sage: os.system('rm ' + tempname)
0
```

save_image (*filename=None*, **args*, ***kwds*)

Save an image representation of self. The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`. Currently this is implemented by calling the `save()` method of self, passing along all arguments and keywords.

Note: Not all image types are necessarily implemented for all graphics types. See `save()` for more details.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((1,1,11), 1,(1,1,1))
sage: q.texture('s')
sage: q.sphere((0,-1,1),1,'s')
sage: tempname = tmp_filename()
sage: q.save_image(tempname)
```

TESTS:

`save_image()` is used for generating animations:

```
sage: def tw_cubic(t):
....:     q = Tachyon()
....:     q.light((1,1,11), 1,(1,1,1))
....:     q.texture('s')
....:     for i in srange(-1,t,0.05):
....:         q.sphere((i,i^2-0.5,i^3), 0.1, 's')
....:     return q

sage: a = animate([tw_cubic(t) for t in srange(-1,1,.3)])
sage: a
Animation with 7 frames
sage: a.show() # optional -- ImageMagick
```

show (*verbose=0*, *extra_opts=''*)

Creates a PNG file of the scene.

EXAMPLES:

```
sage: q = Tachyon()
sage: q.light((-1,-1,10), 1,(1,1,1))
sage: q.texture('s')
sage: q.sphere((0,0,0),1,'s')
sage: q.show(verbose=False)
```

smooth_triangle(*vertex_1*, *vertex_2*, *vertex_3*, *normal_1*, *normal_2*, *normal_3*, *texture*)

Creates a triangle along with a normal vector for smoothing.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.light((1,1,1),.1,(1,1,1))
sage: t.texture('s')
sage: t.smooth_triangle([0,0,0],[0,0,1],[0,1,0],[0,1,1],[-1,1,2],[3,0,0],'s')
sage: t._objects[2].get_vertices()
([0, 0, 0], [0, 0, 1], [0, 1, 0])
sage: t._objects[2].get_normals()
([0, 1, 1], [-1, 1, 2], [3, 0, 0])
```

sphere(*center*, *radius*, *texture*)

Creates the scene information for a sphere with the given center, radius, and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('sphere_texture')
sage: t.sphere((1,2,3), .1, 'sphere_texture')
sage: t._objects[1].str()
'\n        sphere center  1.0 2.0 3.0  rad 0.1 sphere_texture\n        '
```

str()

Returns the complete tachyon scene file as a string.

EXAMPLES:

```
sage: t = Tachyon(xres=500,yres=500, camera_center=(2,0,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t3', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t4', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: t.sphere((0,0.5,0), 0.2, 't2')
sage: t.sphere((0.5,0,0), 0.2, 't3')
sage: t.sphere((0,0,0.5), 0.2, 't4')
sage: t.str().find('PLASTIC')
567
```

texfunc(*type=0*, *center=(0, 0, 0)*, *rotate=(0, 0, 0)*, *scale=(1, 1, 1)*)

INPUT:

- `type` - (default: 0)

  0.No special texture, plain shading

  1.3D checkerboard function, like a rubik's cube

  2.Grit Texture, randomized surface color

  3.3D marble texture, uses object's base color

  4.3D wood texture, light and dark brown, not very good yet

5.3D gradient noise function (can't remember what it looks like)

6.Don't remember

7.Cylindrical Image Map, requires ppm filename (don't know how to specify name in sage?!)

8.Spherical Image Map, requires ppm filename (don't know how to specify name in sage?!)

9.Planar Image Map, requires ppm filename (don't know how to specify name in sage?!)

- `center` - (default: (0,0,0))
- `rotate` - (default: (0,0,0))
- `scale` - (default: (1,1,1))

EXAMPLES: We draw an infinite checkboard:

```
sage: t = Tachyon(camera_center=(2,7,4), look_at=(2,0,0))
sage: t.texture('black', color=(0,0,0), texfunc=1)
sage: t.plane((0,0,0),(0,0,1),'black')
sage: t.show()
```

texture (*name*, *ambient=0.2*, *diffuse=0.8*, *specular=0.0*, *opacity=1.0*, *color=(1.0*, *0.0*, *0.5)*, *texfunc=0*, *phong=0*, *phongsize=0.5*, *phongtype='PLASTIC'*)

INPUT:

- `name` - string; the name of the texture (to be used later)
- `ambient` - (default: 0.2)
- `diffuse` - (default: 0.8)
- `specular` - (default: 0.0)
- `opacity` - (default: 1.0)
- `color` - (default: (1.0,0.0,0.5))
- `texfunc` - (default: 0); a texture function; this is either the output of self.texfunc, or a number between 0 and 9, inclusive. See the docs for self.texfunc.
- `phong` - (default: 0)
- `phongsize` - (default: 0.5)
- `phongtype` - (default: "PLASTIC")

EXAMPLES:

We draw a scene with 4 spheres that illustrates various uses of the texture command:

```
sage: t = Tachyon(camera_center=(2,5,4), look_at=(2,0,0), raydepth=6)
sage: t.light((10,3,4), 1, (1,1,1))
sage: t.texture('mirror', ambient=0.05, diffuse=0.05, specular=.9, opacity=0.9, color=(.8,.8
sage: t.texture('grey', color=(.8,.8,.8), texfunc=3)
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.sphere((4,-1,1), 1, 'mirror')
sage: t.sphere((0,-1,1), 1, 'mirror')
sage: t.sphere((2,-1,1), 0.5, 'mirror')
sage: t.sphere((2,1,1), 0.5, 'mirror')
sage: show(t)  # known bug (:trac:'7232')
```

texture_recolor (*name*, *colors*)

Recolors default textures.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: q = t.texture_recolor('s',[(0,0,1)])
sage: t._objects[1]._color
(0, 0, 1)
```

triangle(*vertex_1*, *vertex_2*, *vertex_3*, *texture*)

Creates a triangle with the given vertices and texture.

EXAMPLES:

```
sage: t = Tachyon()
sage: t.texture('s')
sage: t.triangle([1,2,3],[4,5,6],[7,8,10],'s')
sage: t._objects[1].get_vertices()
([1, 2, 3], [4, 5, 6], [7, 8, 10])
```

class sage.plot.plot3d.tachyon.**TachyonSmoothTriangle**(*a*, *b*, *c*, *da*, *db*, *dc*, *color=0*)

Bases: sage.plot.plot3d.tri_plot.SmoothTriangle

A triangle along with a normal vector, which is used for smoothing.

str()

Returns the scene string for a smoothed triangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonSmoothTriangle
sage: t = TachyonSmoothTriangle([-1,-1,-1],[0,0,0],[1,2,3],[1,0,0],[0,1,0],[0,0,1])
sage: t.str()
'\n        STRI V0  ...  1.0 0.0 0.0  N1  0.0 1.0 0.0   N2  0.0 0.0 1.0 \n            0\n
```

class sage.plot.plot3d.tachyon.**TachyonTriangle**(*a*, *b*, *c*, *color=0*)

Bases: sage.plot.plot3d.tri_plot.Triangle

Basic triangle class.

str()

Returns the scene string for a triangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangle
sage: t = TachyonTriangle([-1,-1,-1],[0,0,0],[1,2,3])
sage: t.str()
'\n        TRI V0  -1.0 -1.0 -1.0   V1  0.0 0.0 0.0    V2  1.0 2.0 3.0 \n            0\n
```

class sage.plot.plot3d.tachyon.**TachyonTriangleFactory**(*tach*, *tex*)

Bases: sage.plot.plot3d.tri_plot.TriangleFactory

A class to produce triangles of various rendering types.

get_colors(*list*)

Returns a list of color labels.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttf.get_colors([1])
['SAGETEX1_0']
```

smooth_triangle(*a*, *b*, *c*, *da*, *db*, *dc*, *color=None*)

Creates a TachyonSmoothTriangle.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttfst = ttf.smooth_triangle([0,0,0],[1,0,0],[0,0,1],[1,1,1],[1,2,3],[-1,-1,2])
sage: ttfst.str()
'\n        STRI V0  0.0 0.0 0.0  ...'
```

triangle(*a*, *b*, *c*, *color=None*)

Creates a TachyonTriangle with vertices a, b, and c.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import TachyonTriangleFactory
sage: t = Tachyon()
sage: t.texture('s')
sage: ttf = TachyonTriangleFactory(t, 's')
sage: ttft = ttf.triangle([1,2,3],[3,2,1],[0,2,1])
sage: ttft.str()
'\n        TRI V0  1.0 2.0 3.0   V1  3.0 2.0 1.0    V2  0.0 2.0 1.0 \n            s\n
```

class sage.plot.plot3d.tachyon.**Texfunc**(*type=0*, *center=(0, 0, 0)*, *rotate=(0, 0, 0)*, *scale=(1, 1, 1)*)

Creates a texture function.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texfunc
sage: t = Texfunc()
sage: t._type
0
```

str()

Returns the scene string for this texture function.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texfunc
sage: t = Texfunc()
sage: t.str()
'0 center  0.0 0.0 0.0  rotate  0.0 0.0 0.0  scale  1.0 1.0 1.0 '
```

class sage.plot.plot3d.tachyon.**Texture**(*name*, *ambient=0.2*, *diffuse=0.8*, *specular=0.0*, *opacity=1.0*, *color=(1.0, 0.0, 0.5)*, *texfunc=0*, *phong=0*, *phongsize=0*, *phongtype='PLASTIC'*)

Stores texture information.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t = Texture('w')
sage: t.str().split()[2:6]
['ambient', '0.2', 'diffuse', '0.8']
```

recolor(*name*, *color*)

Returns a texture with the new given color.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t2 = Texture('w')
sage: t2w = t2.recolor('w2', (.1,.2,.3))
sage: t2ws = t2w.str()
sage: color_index = t2ws.find('color')
sage: t2ws[color_index:color_index+20]
'color  0.1 0.2 0.3  '
```

str()

Returns the scene string for this texture.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import Texture
sage: t = Texture('w')
sage: t.str().split()[2:6]
['ambient', '0.2', 'diffuse', '0.8']
```

sage.plot.plot3d.tachyon.**tostr**(*s*, *length=3*, *out_type=<type 'float'>*)

Converts vector information to a space-separated string.

EXAMPLES:

```
sage: from sage.plot.plot3d.tachyon import tostr
sage: tostr((1,1,1))
' 1.0 1.0 1.0 '
sage: tostr('2 3 2')
'2 3 2'
```

CHAPTER

THIRTEEN

TEXTURE SUPPORT

This module provides texture/material support for 3D Graphics objects and plotting. This is a very rough common interface for Tachyon, x3d, and obj (mtl). See `Texture` and `Texture_class` for full details about options and use.

Initially, we have no textures set:

```
sage: sage.plot.plot3d.base.Graphics3d().texture_set()
set([])
```

However, one can access these textures in the following manner:

```
sage: G = tetrahedron(color='red') + tetrahedron(color='yellow') + tetrahedron(color='red', opacity=0
sage: [t for t in G.texture_set() if t.color == colors.red] # we should have two red textures
[Texture(texture..., red, ff0000), Texture(texture..., red, ff0000)]
sage: [t for t in G.texture_set() if t.color == colors.yellow] # ...and one yellow
[Texture(texture..., yellow, ffff00)]
```

And the Texture objects keep track of all their data:

```
sage: T = tetrahedron(color='red', opacity=0.5)
sage: t = T.get_texture()
sage: t.opacity
0.500000000000000
sage: T # should be translucent
```

AUTHOR:

- Robert Bradshaw (2007-07-07) Initial version.

`sage.plot.plot3d.texture.`**`Texture`**(*id=None*, ***kwds*)

Return a texture.

INPUT:

- `id` - a texture (optional, default: None), a dict, a color, a str, a tuple, None or any other type acting as an ID. If `id` is None, then it returns a unique texture object.
- `texture` - a texture
- `color` - tuple or str, (optional, default: (.4, .4, 1))
- `opacity` - number between 0 and 1 (optional, default: 1)
- `ambient` - number (optional, default: 0.5)
- `diffuse` - number (optional, default: 1)

- `specular` - number (optional, default: 0)
- `shininess` - number (optional, default: 1)
- `name` - str (optional, default: None)
- `**kwds` - other valid keywords

OUTPUT:

A texture object.

EXAMPLES:

Texture from integer `id`:

```
sage: from sage.plot.plot3d.texture import Texture
sage: Texture(17)
Texture(17, 6666ff)
```

Texture from rational `id`:

```
sage: Texture(3/4)
Texture(3/4, 6666ff)
```

Texture from a dict:

```
sage: Texture({'color':'orange','opacity':0.5})
Texture(texture..., orange, ffa500)
```

Texture from a color:

```
sage: c = Color('red')
sage: Texture(c)
Texture(texture..., ff0000)
```

Texture from a valid string color:

```
sage: Texture('red')
Texture(texture..., red, ff0000)
```

Texture from a non valid string color:

```
sage: Texture('redd')
Texture(redd, 6666ff)
```

Texture from a tuple:

```
sage: Texture((.2,.3,.4))
Texture(texture..., 334c66)
```

Textures using other keywords:

```
sage: Texture(specular=0.4)
Texture(texture..., 6666ff)
sage: Texture(diffuse=0.4)
Texture(texture..., 6666ff)
sage: Texture(shininess=0.3)
Texture(texture..., 6666ff)
sage: Texture(ambiant=0.7)
Texture(texture..., 6666ff)
```

class sage.plot.plot3d.texture.**Texture_class**(*id*, *color=(0.4, 0.4, 1)*, *opacity=1*, *ambient=0.5*, *diffuse=1*, *specular=0*, *shininess=1*, *name=None*, ***kwds*)

Bases: sage.structure.sage_object.SageObject

Construction of a texture.

See documentation of Texture for more details and examples.

EXAMPLES:

We create a translucent texture:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t
Texture(texture..., 6666ff)
sage: t.opacity
0.600000000000000
sage: t.jmol_str('obj')
'color obj translucent 0.4 [102,102,255]'
sage: t.mtl_str()
'newmtl texture...\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\nNs 1\nd 0.600000000
sage: t.x3d_str()
"<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1' specularColor='0.0 0.0 0.0'/></A
```

hex_rgb()

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: Texture('red').hex_rgb()
'ff0000'
sage: Texture((1, .5, 0)).hex_rgb()
'ff7f00'
```

jmol_str(*obj*)

Converts Texture object to string suitable for Jmol applet.

INPUT:

- obj - str

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.jmol_str('obj')
'color obj translucent 0.4 [102,102,255]'

sage: sum([dodecahedron(center=[2.5*x, 0, 0], color=(1, 0, 0, x/10)) for x in range(11)]).sh
```

mtl_str()

Converts Texture object to string suitable for mtl output.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.mtl_str()
'newmtl texture...\nKa 0.2 0.2 0.5\nKd 0.4 0.4 1.0\nKs 0.0 0.0 0.0\nillum 1\nNs 1\nd 0.60000
```

tachyon_str()

Converts Texture object to string suitable for Tachyon ray tracer.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.tachyon_str()
'Texdef texture...\n  Ambient 0.333333333333 Diffuse 0.666666666667 Specular 0.0 Opacity 0.6
```

x3d_str()

Converts Texture object to string suitable for x3d.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import Texture
sage: t = Texture(opacity=0.6)
sage: t.x3d_str()
"<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1' specularColor='0.0 0.0 0.0'/
```

sage.plot.plot3d.texture.**is_Texture**(*x*)

Return whether x is an instance of Texture_class.

EXAMPLES:

```
sage: from sage.plot.plot3d.texture import is_Texture, Texture
sage: t = Texture(0.5)
sage: is_Texture(t)
True

sage: is_Texture(4)
False
```

sage.plot.plot3d.texture.**parse_color**(*info*, *base=None*)

Parses the color.

It transforms a valid color string into a color object and a color object into an RBG tuple of length 3. Otherwise, it multiplies the info by the base color.

INPUT:

- info - color, valid color str or number
- base - tuple of length 3 (optional, default: None)

OUTPUT:

A tuple or color.

EXAMPLES:

From a color:

```
sage: from sage.plot.plot3d.texture import parse_color
sage: c = Color('red')
sage: parse_color(c)
(1.0, 0.0, 0.0)
```

From a valid color str:

```
sage: parse_color('red')
RGB color (1.0, 0.0, 0.0)
sage: parse_color('#ff0000')
RGB color (1.0, 0.0, 0.0)
```

From a non valid color str:

```
sage: parse_color('redd')
Traceback (most recent call last):
...
ValueError: unknown color 'redd'
```

From an info and a base:

```
sage: opacity = 10
sage: parse_color(opacity, base=(.2,.3,.4))
(2.0, 3.0, 4.0)
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

p

INDEX

F

G

H

I

J

L

M

O

P

R

S

T

U

V

X