PyVMF

Mar 23, 2020

Contents:

CHAPTER 1

PyVMF

class AllowedVerts(*matrix*, *dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'allowed_verts'

class Alphas(*matrix*, *dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'alphas'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type `dict`, `dict`

class Box(*dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'box'

class Camera(*dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'camera'

class Cameras(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'cameras'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type `list` of *Common* instances

class Color(*r: int = 255*, *g: int = 255*, *b: int = 255*)
Bases: object

Simple RGB color class

Parameters

- **r** (int) – Value for RED between 0 and 255
- **g** (int) – Value for GREEN between 0 and 255
- **b** (int) – Value for BLUE between 0 and 255

export() → Tuple[int, int, int]

random()
Sets a random color

set(*r: int = -1*, *g: int = -1*, *b: int = -1*)
Sets the color

Parameters

- **r** (int) – Value for RED between 0 and 255, if equals to -1 keeps previous value
- **g** (int) – Value for GREEN between 0 and 255, if equals to -1 keeps previous value
- **b** (int) – Value for BLUE between 0 and 255, if equals to -1 keeps previous value

class ColorLight(*r: int = 255*, *g: int = 255*, *b: int = 255*, *brightness: int = 200*)
Bases: *PyVMF.Color*

Simple RGB color class with brightness (used for lights)

Parameters

- **r** (int) – Value for RED between 0 and 255
- **g** (int) – Value for GREEN between 0 and 255
- **b** (int) – Value for BLUE between 0 and 255
- **brightness** (int) – Value for brightness, above 0

export() → Tuple[int, int, int, int]

set_brightness(*brightness: int*)

Parameters **brightness** (int) – New brightness value

class Common
Bases: object

The parent class to all VMF classes that need to be exported to the .VMF file.

ID = 0

copy()
Copies the class using deepcopy()

Returns A deepcopy of itself

Return type *Common* instance

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

ids()

class Connections(*dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'connections'

class Convert
Bases: object

Converts strings to usable instances

static string_to_3x_vertex(*string: str*) → List[Vertex, Vertex, Vertex]

static string_to_color(*string: str*) → PyVMF.Color

static string_to_color_light(*string: str*) → PyVMF.ColorLight

static string_to_uvaxis(*string: str*) → PyVMF.UVaxis

static string_to_vertex(*string: str*) → PyVMF.Vertex

class Cordon(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'cordon'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class Cordons(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'cordons'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class DispInfo(*dic: Optional[dict] = None*, *children: Optional[list] = None*, *parent_side: Optional[PyVMF.Side] = None*)
Bases: *PyVMF.Common*

Keeps track of all the different displacement settings and values

NAME = 'dispinfo'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

is_flipped() → bool
Finds out if the displacement has been flipped, might not work if the face is non rectangular

Returns If the displacement has been flipped

Return type bool

power = None
The displacement power, can only be 2, 3 or 4

startposition

class DispVert
Bases: *PyVMF.Common*

Keeps track of each individual displacement vertex

set(*normal: PyVMF.Vertex*, *distance: int*)
Sets the normal direction and distance

Parameters

- **normal** (*Vertex*) – The normal direction (x, y and z)
- **distance** (int) – How far to go in the normal direction

set_alpha(*amount: int*)
Sets the alpha, used by blend textures, 0 is the first texture, 255 is the second texture, 127 is both

Parameters **amount** (int) – The alpha amount, between 0 and 255

class Distances(*matrix*, *dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'distances'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

class Editor(*dic: Optional[dict] = None*, *parent_type=None*)
Bases: *PyVMF.Common*

NAME = 'editor'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

has_visgroup() → bool

class Entity(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'entity'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class EntityGenerator

Bases: object

Generates entities from scratch, remember you still need to add them to *VMF* using *add_entities()*

static info_decal(*origin: PyVMF.Vertex*, *texture: str*) → PyVMF.InfoDecal

Generates a basic decal

Parameters

- **origin** (*Vertex*) – The position of the decal in the world
- **texture** (str) – The name of the texture (ex: "tools/toolsnodraw")

Returns A generated decal

Return type *InfoDecal*

static info_overlay(*origin: PyVMF.Vertex*, *texture: str*, *angle: PyVMF.Vertex = <PyVMF.Vertex object>*, **sides*) → PyVMF.InfoOverlay

Generates a basic overlay

Parameters

- **origin** (*Vertex*) – The position of the overlay in the world
- **texture** (str) – The name of the texture (ex: "tools/toolsnodraw")
- **angle** (*Vertex*) – The rotation of the overlay in the world

Returns A generated overlay

Return type *InfoOverlay*

static light(*origin: PyVMF.Vertex*, *color: PyVMF.Color*, *brightness: int = 200*) → PyVMF.Light

Generates a basic light

Parameters

- **origin** (*Vertex*) – The position of the light in the world
- **color** (*Color*) – The color of the light
- **brightness** (int) – The brightness of the light

Returns A generated light

Return type *Light*

static prop_static(*origin: PyVMF.Vertex*, *model: str*, *angle: PyVMF.Vertex = <PyVMF.Vertex object>*, *color: PyVMF.Color = <PyVMF.Color object>*, *scale: int = 1*) → PyVMF.PropStatic

Generates a basic prop static

Parameters

- **origin** (*Vertex*) – The position of the prop in the world
- **model** (str) – The name of the prop (ex: models/penguin/penguin.mdl)
- **angle** (*Vertex*) – The rotation of the prop in the world
- **color** (*Color*) – Color of the prop
- **scale** (int) – Size of the prop

Returns A generated prop static

Return type *PropStatic*

class Group(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'group'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class Hidden(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'hidden'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class InfoDecal(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Entity*

SUBNAME = 'infodecal'

class InfoOverlay(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Entity*

SUBNAME = 'info_overlay'

add_sides(**sides*)

class Light(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Entity*

SUBNAME = 'light'

class Matrix(*size: int*)
Bases: *PyVMF.Common*

A grid for keeping track of the displacement values.

Parameters **size** (int) – The size of the 2 dimensional grid

column(*x: int*) → List[DispVert, ...]

Parameters **x** (int) – The column to get

Returns All the disp verts on the given column

Return type list of *DispVert*

export_attr(*attribute*)
Exports the data in .VMF file ready format, used when exporting the PyVMF, you shouldn't need to use this

Parameters **attribute** (str) – Which of the attributes to export (normals, distances, ...)

Returns Row to values association

Return type dict of str: str

extract_dic(*dic*, *a_var=1*, *triangle=False*)
Extracts the data from the .VMF file string, you shouldn't need to use this

Parameters

- **dic** (dict of str: str) – Holds all the rows
- **a_var** (int) – How many variables to group, use 3 to group the 'x y z' format, if single int use 1
- **triangle** (bool) – *TriangleTags* holds 1 less value than all other displacement variables

Returns The x and y position in the matrix and the values

Return type int, int, list of str

get (*x: int*, *y: int*) → PyVMF.DispVert

Parameters

- **x** (int) – Position x in the matrix
- **y** (int) – Position y in the matrix

Returns Displacement information at the given position

Return type *DispVert*

inv_rect (*x*, *y*, *w*, *h*, *step*)

rect (*x: int*, *y: int*, *w: int*, *h: int*) → Generator[DispVert, ...]

Parameters

- **x** – Position x in the matrix
- **y** – Position y in the matrix
- **w** – Width of the rectangle
- **h** – Height of the rectangle

Returns Yields all the disp verts inside the given rectangle

Return type generator of *DispVert*

row (*y: int*) → List[DispVert, ...]

Parameters **y** (int) – The row to get

Returns All the disp verts on the given row

Return type list of *DispVert*

class Normals (*matrix*, *dic: Optional[dict] = None*)

Bases: *PyVMF.Common*

NAME = 'normals'

export ()

Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

class OffsetNormals (*matrix*, *dic: Optional[dict] = None*)

Bases: *PyVMF.Common*

NAME = 'offset_normals'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

class Offsets(*matrix*, *dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'offsets'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

class PropStatic(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Entity*

SUBNAME = 'prop_static'

class Side(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

Corresponds to a face/side of a solid. Sides are defined by 3 vertices, the combination of which define an infinitely large plane, source calculates the intersection between these planes to determine where the edges are. This is not currently calculated in PyVMF, so some methods may behave unpredictably.

Parameters

- **dic** (dict) – All the values to be initialized, if empty default values are used.
- **children** (list) – Holds a potential displacement *DispInfo* to be initialized

NAME = 'side'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

flip(*x=None*, *y=None*, *z=None*)

get_displacement() → PyVMF.DispInfo

Returns The current displacement, only 1 per side

Return type *DispInfo* or None

get_naive_rotation() → int
Gets the rotation if and only if it's a multiple of 90, please don't use this if it's not necessary

Returns Rotation of the face either 0, 90, 180 or 270

Return type int

get_vector()

get_vertices() → List[Vertex, Vertex, Vertex]

Returns All 3 vertices that define the plane

Return type list of *Vertex*

move(*x*, *y*, *z*)
Moves the side by the given amount

Parameters

- **x** (int or float) – Amount to move the x axis by
- **y** (int or float) – Amount to move the y axis by
- **z** (int or float) – Amount to move the z axis by

remove_displacement()
Removes the diplacement from the side

rotate_x(*center: PyVMF.Vertex*, *angle*)
Rotates the side around the x axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_y(*center: PyVMF.Vertex*, *angle*)
Rotates the vertex around the y axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_z(*center: PyVMF.Vertex*, *angle*)
Rotates the vertex around the z axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

set_texture(*new_material: str*)
Sets the given texture on all sides

Parameters **new_material** (str) – The texture to use

class Solid(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

Corresponds to an individual solid just like in Hammer

Parameters

- **dic** (dict) – All the values to be initialized, if empty default values are used.
- **children** (list) – The *Side*'s and *Editor* to be initialized

NAME = 'solid'

add_sides(**args*)
Adds sides to the solid, note that no checks are made for validity

Parameters **args** (list of *Side*) – List of sides to be added

center
Finds the center of the solid based on the average of all vertices. **Can behave unpredictably** as faces only consists of 3 verticies so the center might be off by a tiny amount For a more reliable option see *center_geo()*

Returns The average center of the solid

Return type *Vertex*

center_geo
Finds the center of the solid based on the extremities of all 3 axes. More reliable than *center()*

Returns The geometric center of the solid

Return type *Vertex*

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

flip(*x=None*, *y=None*, *z=None*)

get_3d_extremity(*x: bool = None*, *y: bool = None*, *z: bool = None*) → Tuple[Vertex, List[Vertex, ...]]
Finds the vertices that are the furthest on the given axes, as well as ties

Parameters

- **x** (bool) – False for negative side of the axis, True for positive side
- **y** (bool) – False for negative side of the axis, True for positive side
- **z** (bool) – False for negative side of the axis, True for positive side

Returns The vertex furthest most on the given axes, and the ties, **the champion vertex is included**

Return type *Vertex*, list of *Vertex*

get_all_vertices() → List[Vertex, ...]
Finds all vertices on the solid, including overlapping ones from the different sides, for only unique vertices use *get_only_unique_vertices()*

Returns All the vertices on the solid

Return type list of *Vertex*

get_axis_extremity(*x: Optional[bool] = None*, *y: Optional[bool] = None*, *z: Optional[bool] = None*) → PyVMF.Vertex
Finds the vertex that is the furthest on the given axis, **only 1 axis per method call**, see *get_3d_extremity()*

Parameters

- **x** (bool) – False for negative side of the axis, True for positive side
- **y** (bool) – False for negative side of the axis, True for positive side
- **z** (bool) – False for negative side of the axis, True for positive side

Returns The vertex the furthest most on the given axis

Return type *Vertex*

get_displacement_matrix_sides() → List[Matrix, ...]
Gets the matrices from all the sides that have displacements, use *get_displacement_sides()* to get the sides instead

Returns The matrices from the sides with displacements on them

Type list of *Matrix*

get_displacement_sides() → List[PyVMF.Side]
Gets the sides that have displacements, use *get_displacement_matrix_sides()* to get the matrices directly instead

Returns The sides with displacements on them

Return type list of *Side*

get_linked_vertices(*vertex: Vertex*, *similar=0.0*) → List[Vertex, ...]

Parameters

- **vertex** (*Vertex*) – The vertex to check against
- **similar** (float) – Distance between vertices to be considered similar (in Hammer units)

Returns All vertices that are in close proximity to the given vertex itself included

Return type list of *Vertex*

get_only_unique_vertices() → List[Vertex, ...]
Finds all unique vertices on the solid, **you should not use this for vertex manipulation as changing one doesn't change all of them**. See *get_all_vertices()*

Returns all unique vertices

Return type list of *Vertex*

get_sides() → List[Side, ...]

Returns All the sides on the solid

Return type list of *Side*

get_texture_sides(*name: str*, *exact=False*) → List[Side, ...]

Parameters

- **name** (string) – The name of the texture including path (ex: tools/toolsnodraw)
- **exact** (bool) – Determines if the material has to be letter for letter the same or just contain the string

Returns The sides using the given texture

Return type list of *Side*

has_texture(*name: str*, *exact=False*) → bool

Parameters

- **name** (string) – The name of the texture including path (ex: tools/toolsnodraw)
- **exact** (bool) – Determines if the material has to be letter for letter the same or just contain the string

Returns if any sides of the solid contain the given texture

Return type bool

is_simple_solid() → bool

Returns A solid is considered simple if it has 6 or less sides

Return type bool

link_vertices(*similar=0.0*)
Tries to link all the vertices that are similiar

Parameters **similar** –

move(*x*, *y*, *z*)
Moves all sides of the solid by the given amount in Hammer units

Parameters

- **x** (int or float) –
- **y** (int or float) –
- **z** (int or float) –

naive_subdivide(*x=1*, *y=1*, *z=1*) → List[Solid, ...]
Naively subdivides a copy of the solid, works best for rectangular shapes. It's naive because it scales down the solid then creates an array from that

Parameters

- **x** (int) – Amount of cuts on the x axis
- **y** (int) – Amount of cuts on the y axis
- **z** (int) – Amount of cuts on the z axis

Returns Solids from a subdivided solid

Return type list of *Solid*

remove_all_displacements()
Removes all displacements from the solid

replace_texture(*old_material: str*, *new_material: str*)
Checks all the sides if they have the given texture, if so replace it

Parameters

- **old_material** (String) – The texture to check
- **new_material** (String) – The texture to replace the old one with

rotate_x(*center: PyVMF.Vertex*, *angle*)
Rotates the solid around the x axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_y(*center: PyVMF.Vertex*, *angle*)
Rotates the solid around the y axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_z (*center: PyVMF.Vertex*, *angle*)
Rotates the solid around the z axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

scale (*center: PyVMF.Vertex*, *x=1.0*, *y=1.0*, *z=1.0*)
Scales the solid using ratios. For example using the center of the solid and values of 2 makes it twice as big

Parameters

- **center** (*Vertex*) – The point from which the scaling is based, use the center of the solid for traditional scaling
- **x** (int or float) – Scale ratio on the x axis
- **y** (int or float) – Scale ratio on the y axis
- **z** (int or float) – Scale ratio on the z axis

set_texture (*new_material: str*)
Sets the given texture on all sides

Parameters **new_material** (str) – The texture to replace them all

size

Returns The total size of the bounding rectangle around the solid

Return type *Vertex*

window (*direction: Vertex = None*) → List[Solid, Solid, Solid, Solid]
Creates a hole in the wall, only works on 90 degree blocks

Parameters **direction** (*Vertex*) – If set defines the direction the hole will be made, requires exactly 2 non-zero values

Returns The 4 blocks surrounding the hole

Return type list of *Solid*

class SolidGenerator
Bases: object

Generates solids from scratch, remember you still need to add them to *VMF* using *add_solids()*

static cube (*vertex: PyVMF.Vertex*, *w*, *h*, *l*, *center=False*, *dev=0*) → PyVMF.Solid
Generates a solid cube

Parameters

- **vertex** (*Vertex*) – Start position from which to build the cube
- **w** (int) – Width of the cube
- **h** (int) – Height of the cube
- **l** (int) – Length of the cube
- **center** (bool) – If set to True centers the solid on the vertex
- **dev** (int) – If set, changes the cube texture, see *dev_material()*

Returns A generated solid

Return type *Solid*

static dev_material(*solid: PyVMF.Solid*, *dev: int*)
Changes the material of the solid to single color dev textures, quick and useful when testing

Parameters

- **solid** (*Solid*) – The target solid
- **dev** (int) – The target texture between 1 and 5

static displacement_triangle(*vertex: PyVMF.Vertex*, *w*, *h*, *l*, *dev=0*) → PyVMF.Solid
Generates a displacement triangle (L shaped viewed from above)

Parameters

- **vertex** (*Vertex*) – Start position from which to build the triangle
- **w** (int) – Width of the triangle
- **h** (int) – Height of the triangle
- **l** (int) – Length of the triangle
- **dev** (int) – If set, changes the triangle texture, see *dev_material()*

Returns A generated triangle

Return type *Solid*

static room(*vertex: Vertex*, *w*, *h*, *l*, *thick: int = 64*, *dev=0*) → List[Solid, Solid, Solid, Solid, Solid, Solid]
Generates a sealed cubed room

Parameters

- **vertex** (*Vertex*) – Center position of the room
- **w** (int) – Width of the room
- **h** (int) – Height of the room
- **l** (int) – Length of the room
- **thick** (int) – The thickness of the walls
- **dev** (int) – If set, changes the room texture, see *dev_material()*

Returns A generated room

Return type list of *Solid*

static surf_ramp(*vertex: PyVMF.Vertex*, *w*, *h*, *l*, *top_cut=32*, *side_cut=32*, *center=False*, *ramp_texture='cs_italy/cobble02'*, *top_texture='cs_italy/plasterwall02a'*, *side_texture='cs_italy/plasterwall02a'*) → PyVMF.Solid
Generates a ramp (triangle viewed from above:)

Parameters

- **vertex** (*Vertex*) – Start position from which to build the ramp (bottom middle of the ramp)
- **w** (int) – Width of the ramp
- **h** (int) – Height of the ramp
- **l** (int) – Length of the ramp
- **top_cut** (int) – Width of the cut at the top

- **side_cut** (int) – Height of the cut on the side
- **center** (bool) – If set to True centers the ramp on the vertex
- **ramp_texture** (str) –
- **top_texture** (str) –
- **side_texture** (str) –

Returns A generated ramp

Return type *Solid*

class TriangleTag(*x*, *y*)
Bases: *PyVMF.Common*

class TriangleTags(*matrix*, *dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'triangle_tags'

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

class UVaxis(*x*, *y*, *z*, *offset*, *scale*)
Bases: *PyVMF.Common*

export()
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

localize(*side*)

class VMF
Bases: object

Equivalent to a single .VMF file, holds all categories and all sub-categories

add_entities(**args*)
Adds entities to the entity list

Parameters **args** – Entities to add

add_section(*section: importer.TempCategory*)
Adds temporary categories to the VMF, used when reading .VMF files, you shouldn't need to use this

Parameters **section** (*importer.TempCategory*) – The temporary category to add

add_solids(**args*)
Adds solids to the world

Parameters **args** – Solids to add

add_to_visgroup(*name: str*, **args*)
Adds the given elements to a visgroup, if it doesn't exist one is created

Parameters

- **name** (str) – Name of the visgroup

- **args** – Elements to add to the visgroup

blank_vmf()
Generates necessary categories (overwriting existing), use *new_vmf()* to generate the VMF itself

export(*filename: str*)
Exports the VMF to a .VMF file

Parameters **filename** (str) – Exported file name, use a different filename or it will overwrite the existing file

get_all_from_visgroup(*name: str*)
Gets everything from the visgroup

Parameters **name** (str) – Name of the visgroup

Returns Solids and entities in the visgroup

Return type list of (*Entity* or *Solid*)

get_entities(*include_hidden=False*, *include_solid_entities=False*) → List[Entity, ...]
Gets all the entities

Parameters

- **include_hidden** (bool) – Whether to include quick hidden solids (Hammer "h" hotkey) or not
- **include_solid_entities** (bool) – Whether to include solid entities (ex: trigger_teleport) or not

Returns Entities in the VMF

Return type list of *Entity*

get_group_center(*group: list*, *geo=False*) → PyVMF.Vertex
Gets a vertex based on the average center of all the solids

Parameters

- **group** (list of *Solid*) – All the solids to include
- **geo** – Whether to use the geometric center or not, see *center()* and *center_geo()*

Returns The average center position of all the solids

Return type *Vertex*

get_solids(*include_hidden=False*, *include_solid_entities=True*) → List[Solid, ...]
Gets all the solids

Parameters

- **include_hidden** (bool) – Whether to include quick hidden solids (Hammer "h" hotkey) or not
- **include_solid_entities** (bool) – Whether to include solid entities (ex: trigger_teleport) or not

Returns Solids in the VMF

Return type list of *Solid*

get_solids_and_entities(*include_hidden=False*)
Gets all the solids and entities

Parameters **include_hidden** (bool) – Whether to include quick hidden solids (Hammer "h" hotkey) or not

Returns Solids and entities in the VMF

Return type list of (*Entity* or *Solid*)

info_in_console = False

mark_vertex(*vertex: PyVMF.Vertex*, *size: int = 32*, *dev: int = 1*, *visgroup: Optional[str] = None*)
Quickly adds a solid cube at the given vertex, useful for debugging

Parameters

- **vertex** (*Vertex*) – The position on which the cube is centered on
- **size** (int) – The size of the cube
- **dev** (int) – The texture given to the cube, see *dev_material()*
- **visgroup** (None or str) – Optionally adding the cube to an existing visgroup

sort_by_attribute(*category_list: list*, *attr: str*)
Sorts the list based on one of their attributes

Parameters

- **category_list** (list) – All the elements to sort
- **attr** – The attribute to sort by, for example *center_geo.x* for *Solid*

Returns The elements sorted in increasing order

Return type list

class Vector(*x*, *y*, *z*)
Bases: *PyVMF.Common*

angle(*other*)

angle_to_origin()

cross(*other*)

dot(*other*)

mag()

normalize()

to_vertex()

classmethod vector_from_2_vertices(*v1: PyVMF.Vertex*, *v2: PyVMF.Vertex*)

classmethod vectors_from_side(*side: PyVMF.Side*) → Tuple[PyVMF.Vector, PyVMF.Vector]

class VersionInfo(*dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'versioninfo'

class Vertex(*x=0*, *y=0*, *z=0*)
Bases: *PyVMF.Common*

Corresponds to a single position on the Hammer grid

Parameters

- **x** (int or float) – x position

- **y** (int or float) – y position
- **z** (int or float) – z position

align_to_grid()
Turns x, y and z into integers

diff(*other*) → PyVMF.Vertex

Parameters **other** – The vertex to differentiate with

Returns The difference in distance between 2 vertices

Return type *Vertex*

divide(*amount*)
Divides all the axes uniformly by the given amount (for separate division see *divide_separate()*)

Parameters **amount** (int or float) – How much to divide each axis by

divide_separate(*x*, *y*, *z*)
Divides all the axes separatedly by the given amounts (for uniform division see *divide()*)

Parameters

- **x** (int or float) – Amount to divide x axis by
- **y** (int or float) – Amount to divide y axis by
- **z** (int or float) – Amount to divide z axis by

export() → Tuple[int, int, int]
Gets all the variables than need to be exported into the .VMF file

Returns All predefined (in *export_list*) variable names and their associated values

Return type dict, dict

flip(*x=None*, *y=None*, *z=None*)

move(*x*, *y*, *z*)
Moves the vertex by the given amount

Parameters

- **x** (int or float) – Amount to move the x axis by
- **y** (int or float) – Amount to move the y axis by
- **z** (int or float) – Amount to move the z axis by

multiply(*amount*)
Multiplies all the axes uniformly by the given amount

Parameters **amount** (int or float) – How much to multiply each axis by

rotate_x(*center: PyVMF.Vertex*, *angle*)
Rotates the vertex around the x axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_y(*center: PyVMF.Vertex*, *angle*)
Rotates the vertex around the y axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

rotate_z(*center: PyVMF.Vertex*, *angle*)
Rotates the vertex around the z axis

Parameters

- **center** (*Vertex*) – The point to rotate around
- **angle** (int or float) – How much to rotate in degrees

set(*x*, *y*, *z*)
Sets the vertex position to the given position

Parameters

- **x** (int or float) – New x position
- **y** (int or float) – New y position
- **z** (int or float) – New z position

similar(*other*, *accuracy=0.001*) → bool
Compares the current vertex with the given one to see if they are similar

Parameters

- **other** (*Vertex*) –
- **accuracy** (float) – Distance from the current vertex to be considered similar (in Hammer units)

Returns If the given vertex is within the proximity of the current vertex

Return type bool

class ViewSettings(*dic: Optional[dict] = None*)
Bases: *PyVMF.Common*

NAME = 'viewsettings'

class VisGroup(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'visgroup'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

class VisGroups(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'visgroups'

export_children() → Tuple[PyVMF.VisGroup, ...]
Gets all the children classes

Returns All predefined children classes

Return type list of *Common* instances

get_visgroups() → List[PyVMF.VisGroup]

new_visgroup(*name: str*) → PyVMF.VisGroup

class World(*dic: Optional[dict] = None*, *children: Optional[list] = None*)
Bases: *PyVMF.Common*

NAME = 'world'

export_children()
Gets all the children classes

Returns All predefined children classes

Return type `list` of *Common* instances

load_vmf(*name: str*, *merge_vertices=0.0001*) → PyVMF.VMF
Loads a .VMF file

Parameters

- **name** (`str`) – The OS file to open, path needs to be included
- **merge_vertices** (`int` or `float`) – Vertices on a solid within this distance are merged into a single vertex class, set to 0 for no merging

Returns A loaded VMF

Return type *VMF*

new_vmf() → PyVMF.VMF
Generates a VMF with the necessary classes

Returns A blank VMF

Return type *VMF*

VMF()	Equivalent to a single .VMF file, holds all categories and all sub-categories
Solid(dic, children)	Corresponds to an individual solid just like in Hammer
SolidGenerator	Generates solids from scratch, remember you still need to add them to *VMF* using *add_solids()*
Entity(dic, children)	
EntityGenerator	Generates entities from scratch, remember you still need to add them to *VMF* using *add_entities()*
Side(dic, children)	Corresponds to a face/side of a solid.
Vertex([x, y, z])	Corresponds to a single position on the Hammer grid
DispInfo(dic, children, parent_side)	Keeps track of all the different displacement settings and values
DispVert()	Keeps track of each individual displacement vertex
Matrix(size)	A grid for keeping track of the displacement values.

1.1 PyVMF.VMF

class VMF
Equivalent to a single .VMF file, holds all categories and all sub-categories

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
add_entities(*args)	Adds entities to the entity list
add_section(section)	Adds temporary categories to the VMF, used when reading .VMF files, you shouldn't need to use this
add_solids(*args)	Adds solids to the world
add_to_visgroup(name, *args)	Adds the given elements to a visgroup, if it doesn't exist one is created
blank_vmf()	Generates necessary categories (overwriting existing), use *new_vmf()* to generate the VMF itself
export(filename)	Exports the VMF to a .VMF file
get_all_from_visgroup(name)	Gets everything from the visgroup
get_entities([include_hidden, ...])	Gets all the entities
get_group_center(group[, geo])	Gets a vertex based on the average center of all the solids
get_solids([include_hidden, ...])	Gets all the solids
get_solids_and_entities([include_hidden])	Gets all the solids and entities
mark_vertex(vertex, size, dev, visgroup)	Quickly adds a solid cube at the given vertex, useful for debugging
sort_by_attribute(category_list, attr)	Sorts the list based on one of their attributes

Attributes

info_in_console

1.2 PyVMF.Solid

class Solid(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

Corresponds to an individual solid just like in Hammer

Parameters

- **dic** (dict) – All the values to be initialized, if empty default values are used.
- **children** (list) – The *Side*'s and *Editor* to be initialized

__init__(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dic, children)	Initialize self.
add_sides(*args)	Adds sides to the solid, note that no checks are made for validity
copy()	Copies the class using deepcopy()
export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
flip([x, y, z])	

Continued on next page

Table 4 – continued from previous page

get_3d_extremity(x, y, z)	Finds the vertices that are the furthest on the given axes, as well as ties
get_all_vertices()	Finds all vertices on the solid, including overlapping ones from the different sides, for only unique vertices use *get_only_unique_vertices()*
get_axis_extremity(x, y, z)	Finds the vertex that is the furthest on the given axis, **only 1 axis per method call**, see *get_3d_extremity()*
get_displacement_matrix_sides()	Gets the matrices from all the sides that have displacements, use *get_displacement_sides()* to get the sides instead
get_displacement_sides()	Gets the sides that have displacements, use *get_displacement_matrix_sides()* to get the matrices directly instead
get_linked_vertices(vertex[, similar])	**param vertex** The vertex to check against
get_only_unique_vertices()	Finds all unique vertices on the solid, **you should not use this for vertex manipulation as changing one doesn't change all of them**.
get_sides()	**return** All the sides on the solid
get_texture_sides(name[, exact])	**param name** The name of the texture including path (ex: tools/toolsnodraw)
has_texture(name[, exact])	**param name** The name of the texture including path (ex: tools/toolsnodraw)
ids()	
is_simple_solid()	**return** A solid is considered simple if it has 6 or less sides
link_vertices([similar])	Tries to link all the vertices that are similiar
move(x, y, z)	Moves all sides of the solid by the given amount in Hammer units
naive_subdivide([x, y, z])	Naively subdivides a copy of the solid, works best for rectangular shapes.
remove_all_displacements()	Removes all displacements from the solid
replace_texture(old_material, new_material)	Checks all the sides if they have the given texture, if so replace it
rotate_x(center, angle)	Rotates the solid around the x axis
rotate_y(center, angle)	Rotates the solid around the y axis
rotate_z(center, angle)	Rotates the solid around the z axis
scale(center[, x, y, z])	Scales the solid using ratios.
set_texture(new_material)	Sets the given texture on all sides

Continued on next page

Table 4 – continued from previous page

window(direction)	Creates a hole in the wall, only works on 90 degree blocks

Attributes

ID	
NAME	
center	Finds the center of the solid based on the average of all vertices.
center_geo	Finds the center of the solid based on the extremities of all 3 axes.
size	**return** The total size of the bounding rectangle around the solid

1.3 PyVMF.SolidGenerator

class SolidGenerator

Generates solids from scratch, remember you still need to add them to *VMF* using *add_solids()*

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
cube(vertex, w, h, l[, center, dev])	Generates a solid cube
dev_material(solid, dev)	Changes the material of the solid to single color dev textures, quick and useful when testing
displacement_triangle(vertex, w, h, l[, dev])	Generates a displacement triangle (L shaped viewed from above)
room(vertex, w, h, l, thick[, dev])	Generates a sealed cubed room

1.4 PyVMF.Entity

class Entity(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

__init__(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dic, children)	Initialize self.
copy()	Copies the class using deepcopy()

Continued on next page

Table 7 – continued from previous page

export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
ids()	

Attributes

ID
NAME

1.5 PyVMF.EntityGenerator

class EntityGenerator

Generates entities from scratch, remember you still need to add them to *VMF* using *add_entities()*

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
light(origin, color, brightness)	Generates a basic light
prop_static(origin, model, angle, color, scale)	Generates a basic prop static

1.6 PyVMF.Side

class Side(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

Corresponds to a face/side of a solid. Sides are defined by 3 vertices, the combination of which define an infinitely large plane, source calculates the intersection between these planes to determine where the edges are. This is not currently calculated in PyVMF, so some methods may behave unpredictably.

Parameters

- **dic** (dict) – All the values to be initialized, if empty default values are used.
- **children** (list) – Holds a potential displacement *DispInfo* to be initialized

__init__(*dic: Optional[dict] = None*, *children: Optional[list] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dic, children)	Initialize self.
copy()	Copies the class using deepcopy()
export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
flip([x, y, z])	

Continued on next page

Table 10 – continued from previous page

get_displacement()	**return** The current displacement, only 1 per side
get_vector()	
get_vertices()	**return** All 3 vertices that define the plane
ids()	
move(x, y, z)	Moves the side by the given amount
remove_displacement()	Removes the diplacement from the side
rotate_x(center, angle)	Rotates the side around the x axis
rotate_y(center, angle)	Rotates the vertex around the y axis
rotate_z(center, angle)	Rotates the vertex around the z axis

Attributes

ID
NAME

1.7 PyVMF.Vertex

class Vertex(*x=0*, *y=0*, *z=0*)

Corresponds to a single position on the Hammer grid

Parameters

- **x** (int or float) – x position
- **y** (int or float) – y position
- **z** (int or float) – z position

__init__(*x=0*, *y=0*, *z=0*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([x, y, z])	Initialize self.
align_to_grid()	Turns x, y and z into integers
copy()	Copies the class using deepcopy()
diff(other)	**param other** The vertex to differentiate with
divide(amount)	Divides all the axes uniformly by the given amount (for separate division see *divide_separate()*)
divide_separate(x, y, z)	Divides all the axes separatedly by the given amounts (for uniform division see *divide()*)

Continued on next page

Table 12 – continued from previous page

export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
flip([x, y, z])	
ids()	
move(x, y, z)	Moves the vertex by the given amount
multiply(amount)	Multiplies all the axes uniformly by the given amount
rotate_x(center, angle)	Rotates the vertex around the x axis
rotate_y(center, angle)	Rotates the vertex around the y axis
rotate_z(center, angle)	Rotates the vertex around the z axis
set(x, y, z)	Sets the vertex position to the given position
similar(other[, accuracy])	Compares the current vertex with the given one to see if they are similar

Attributes

ID

1.8 PyVMF.DispInfo

class DispInfo(*dic: Optional[dict] = None*, *children: Optional[list] = None*, *parent_side: Optional[PyVMF.Side] = None*)

Keeps track of all the different displacement settings and values

__init__(*dic: Optional[dict] = None*, *children: Optional[list] = None*, *parent_side: Optional[PyVMF.Side] = None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dic, children, parent_side)	Initialize self.
copy()	Copies the class using deepcopy()
export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
ids()	

Attributes

ID
NAME

1.9 PyVMF.DispVert

class DispVert

Keeps track of each individual displacement vertex

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.
copy()	Copies the class using deepcopy()
export()	Gets all the variables than need to be exported into the .VMF file
export_children()	Gets all the children classes
ids()	
set(normal, distance)	Sets the normal direction and distance
set_alpha(amount)	Sets the alpha, used by blend textures, 0 is the first texture, 255 is the second texture, 127 is both

Attributes

ID

1.10 PyVMF.Matrix

class Matrix(*size: int*)
A grid for keeping track of the displacement values.

Parameters **size** (int) – The size of the 2 dimensional grid

__init__(*size: int*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(size)	Initialize self.
column(x)	**param x** The column to get
copy()	Copies the class using deepcopy()
export()	Gets all the variables than need to be exported into the .VMF file
export_attr(attribute)	Exports the data in .VMF file ready format, used when exporting the PyVMF, you shouldn't need to use this
export_children()	Gets all the children classes
extract_dic(dic[, a_var, triangle])	Extracts the data from the .VMF file string, you shouldn't need to use this
get(x, y)	**param x** Position x in the matrix
ids()	

Continued on next page

Table 18 – continued from previous page

inv_rect(x, y, w, h, step)	
rect(x, y, w, h)	**param x** Position x in the matrix
row(y)	**param y** The row to get

Attributes

ID

CHAPTER 2

Tools

num(*s: str*)

Tries to turn string into int or float

Parameters **s** (`str`) – The string to parse

Returns If unable to convert returns the input

Return type `int` or `float` or `str`

CHAPTER 3

Importer

class TempCategory(*category*, *indent*)

Bases: `object`

Temporarily holds the VMF categories when reading the .VMF file

Parameters

- **category** (`str`) – The category name
- **indent** (`int`) – The level of indentation (how far nested inside other classes)

add_child(*category*, *indent*)

Parameters

- **category** (`str`) – The category name
- **indent** (`int`) – The level of indentation (how far nested inside other classes)

add_line(*line*, *indent*)

Parameters

- **line** (`str`) – The line of data to add to the current category
- **indent** (`int`) – The level of indentation (how far nested inside other classes)

clean_up()

Goes through all the data to remove unecessary characters

file_parser(*file*)

Opens the file, extracts data line by line and turns it all into temporary categories

Parameters **file** (`str`) – The OS file to open, path needs to be included

Returns All the top level categories

Return type `list` of *TempCategory*

CHAPTER 4

Obj

CHAPTER 5

Triangulate Displacement

triangulate_displacement(*vmf: PyVMF.VMF, group: List[PyVMF.Solid], base_triangle: PyVMF.Solid = None, resolution=1, height=2*)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

t

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W