Dustmaker

Release 0.4.0

Mark Gordon

Jul 10, 2022

CONTENTS

CHAPTER

ONE

SUBMODULES

CHAPTER

TWO

LEVEL MODULE

Module defining the primary interface for working with levels in dustmaker.

class LevelType(*value*)

Bases: IntEnum

Enum defining the different level types.

NORMAL = 0

NEXUS = 1

NEXUS_MP = 2

KOTH = 3

SURVIVAL = 4

DUSTMOD = 6

class PlayerPosition(*variables: Dict[str,* Variable*]*, *player: int*)

Bases: object

Used internally to manage player position accessors. Meant to be used through accesss to *Level.start_position()*.

x

Player start x-coordinate in pixels

Type
int

y

Player start y-coordinate in pixels

Type
int

class Level(**, parent: Optional[*Level*] = None*)

Bases: object

Represents a Dustforce level/map or the backdrop to its *parent* level. If this is a backdrop then *parent* will be set to the parent *Level* object.

parent

For backdrops this is the containing level. Otherwise this is set to None.

Type
Level, optional

`backdrop`

The backdrop Level object or None if this is a backdrop level. Backdrop levels are scaled up 16x from the parent level's coordinate system and should only contain tiles and props.

Type
Level, optional

`tiles`

A dict mapping (layer, x, y) to Tile objects.

Type
dict[(int, int, int), *Tile*]

`props`

A dict mapping prop ids to (layer, x, y, Prop) tuples.

Type
dict[int, (int, float, float, *Prop*)]

`entities`

A dict mapping entity ids to (x, y, Entity) tuples. Ignored for backdrops.

Type
dict[int, (float, float, *Entity*)]

`variables`

A raw mapping of string keys to Variable objects. Some of these variables have nicer accessor properties like `name` but may be accessed raw through this dictionary. Ignored for backdrops.

Type
dict[str, *Variable*]

`sshot`

The level thumbnail image as a PNG binary. Ignored for backdrops.

Type
bytes

`property name`

Wrapper around *variables['level_name']* of type `VariableString`. Defaults to *b''*.

Type
bytes

`property virtual_character`

Wrapper around *variables['vector_character']* of type `VariableBool`. Defaults to *False*.

Type
bool

`property level_type`

Level type of the level, see `LevelType` enum. Defaults to *0*.

Type
int

property dustmod_version

Wrapper around *variables['dustmod_version']* of type `VariableString`. Defaults to *b''*.

Type
bytes

start_position(*player: int = 1*) → *PlayerPosition*

Access and modify the starting position of each player.

Parameters
player (*int, optional*) – The player to access the starting position of. Valid options are 1, 2, 3, 4. Defaults to player 1.

Returns
An accessor class with *x* and *y* attributes that can be get/set.

add_prop(*layer: int*, *x: float*, *y: float*, *prop:* Prop, *id_num: Optional[int] = None*) → int

Adds a new `Prop` to the level and returns its id. This is the preferred way of adding props to a level. Do not directly add props to the *props* attribute and use this method as it may overwrite props.

Parameters

- **x** (*float*) – The x position of the prop.
- **y** (*float*) – The y position of the prop.
- **prop** (Prop) – The Prop object to add to the level.
- **id_num** – (int, optional): The prop identifier. If not set the identifier will be allocated for you.

Returns
The ID of the newly added prop.

Raises
LevelException – If the given ID is already in use.

add_entity(*x: float*, *y: float*, *entity:* Entity, *id_num: Optional[int] = None*) → int

Adds a new `Entity` to the level and returns its id. This is the preferred way of adding entities to a level. Do not directly add entities to the *entities* attribute and use this method as it may overwrite entities.

Parameters

- **x** (*float*) – The x position of the entity.
- **y** (*float*) – The y position of the entity.
- **entity** (Entity) – The Entity object to add to the level.
- **id_num** – (int, optional): The entity identifier. If not set the identifier will be allocated for you.

Returns
The ID of the newly added entity.

Raises
LevelException – If the given ID is already in use.

translate(*x: float*, *y: float*) → None

Translate (move) the entire level. This is just a convenience method around *transform()*.

Parameters

- **x** (*float*) – The number of pixels to move horizontally.

- **y** (*float*) – The number of pixels to move vertically.

remap_ids(*min_id: int = 100*) → None

Remap prop and entity ids starting at *min_id*. This is useful when attempting to merge two levels to keep their ID space separate. Do not call directly on a backdrop level.

Parameters

min_id (*int, optional*) – The minimum ID to assign to a prop or entity.

> **Warning:** Dustmaker has no way to automatically remap entity IDs in script persist data.

calculate_max_id(*reset: bool = True*) → int

Calculates the maximum prop or entity ID currently in use. This will always return at least 100 due to Dustforce reserving many of the lower IDs for special entities like the camera.

Parameters

reset (*bool, optional*) – If set (the default) the internal next ID allocator will be reset based off this result. Otherwise the max ID will be at least one less than the next ID.

merge(*other_level:* Level, *remap_ids: bool = True*) → None

Merge another level into this one.

Parameters

- **other_map** (Level) – The level to merge into this one.
- **remap_ids** (*bool, optional*) – Wether to remap the ID space of each level so they do not interfere with each other. This is True by default.

transform(*mat: TxMatrix*) → None

Transforms the level with the given affine transformation matrix. Note that this will probably not produce desirable results if the transformation matrix is not some mixture of a translation, flip, and 90 degree rotations. Use *upscale()* if you wish to performan an upscale as well as a transformation.

In most cases you can use one of *flip_horizontal()*, flip_vertical:, :meth:`rotate(), *translate()*, *upscale()* instead of this method.

Parameters

mat – The 3 by 3 affine transformation matrix [x', y', 1]' = mat * [x, y, 1]'. It should be of the form *mat = [[xx, xy, ox], [yx, yy, oy], [0, 0, 1]]*.

> **Warning:** Dustmaker has no way to automatically transform positional data in script persist data.

flip_horizontal() → None

Flips the level horizontally. This is a convenience function around *transform()*.

flip_vertical() → None

Flips the level vertically. This is a convenience function around *transform()*.

rotate(*times: int = 1*) → None

Rotates the level 90 degrees clockwise. This is a convenience function around *transform()*.

Parameters

times (*int, optional*) – The number of 90 degree clockwise rotations to perform. *times* may be negative to perform counterclockwise rotations.

upscale(*factor: int*, *, *mat: TxMatrix = ((1, 0, 0), (0, 1, 0), (0, 0, 1))*) → None

Increase the size of the level along each axis.

Parameters

- **factor** (*int*) – The scaling factor (>1). e.g. if *factor* = 2 each tile tile will be represented by a 2x2 tile squre in the resulting level.
- **mat** (*optional*) – An additional transformation matrix to pass to *transform()* along with the upscaling.

calculate_edge_visibility(*, *visible_callback: Optional[Callable[[int, int,* TileSide*,* Tile*,* Tile*], bool]] = None*) → None

This method will automatically calculate edge solidity and visibility in a way meant to match Dustforce rules.

Solidity will always imply visibility. An edge that doesn't exist for the given tile shape is always not visible. Otherwise an edge that is not flush to a tile border (e.g. the diagonal of a slope tile) is solid. Otherwise if the neighboring side does not exist or is not also flush the edge is solid.

In any other case the edge is not solid. If the tile sprite information matches its neighbor the edge is not visible. Otherwise *visible_callback* is called to determine if the edge is visible. If *visible_callback* is not set this defaults to the edge being visible if it's a bototm or right edge. bottom or right edge.

Parameters

visible_callback (*Callable*) – Callback used to determine if a given edge should be visible if all other checks have passed. Typically this function should be anti-symetric so that there are not overlapping visible edges. Called as *visible_callback(x, y, side, tile, neighbor_tile)*.

calculate_edge_caps() → None

Calculates edge/filth cap flags and angles. This should be called after all edge visibilty has been determined (see *calculate_edge_visibility()*).

To compute edge caps we consider only visible edges. Non-visible edges will have their cap data appropriately zeroed. For each visible edge we consider it in both orientations; going clockwise and counter-clockwise around the tile.

Edges that end between tile widths (i.e. for the slant edge of a slant) can never have an edge cap. For these edges the edge/filth cap should be set to False and the angles zeroed.

The first step to computing the cap flag and angle for a given edge orientation is to find the "joining" edge. A joining edge must have the following properties:

- Belong to a tile with the same sprite
- Be the same side of the tile (i.e. ground edges don't connect to walls)
- Have a starting point equal to our edge's ending point
- Be in the same orientation as our edge

If there are multiple joining edges the one that moves the most "away" from our tile should be selected (when traversing clockwise the edge that goes the most counter-clockwise and vice versa).

If there is no joining edge the edge cap should be set to True and the edge angle should be zeroed. If there is a joining edge the edge cap should be set to False and the edge angle should be half the angle delta rounded down. Clockwise turns should be positive, counter-clockwise turns should be negative.

Finally if the edge has no filth then the filth cap and angle should be zeroed. If there is filth on this edge but not the joining edge, or the filth sprites/spike types don't match, the filth cap should be set to True and filth angle to 0. Otherwise the filth cap should be False and the filth angle should match the edge angle.

CHAPTER

THREE

TILE MODULE

Module defining the tile respresentation in dustmaker.

class TileSpriteSet(*value*)

Bases: IntEnum

Used to describe what set of tiles a tile's sprite comes from.

NONE_0 = 0

MANSION = 1

FOREST = 2

CITY = 3

LABORATORY = 4

TUTORIAL = 5

NEXUS = 6

NONE_7 = 7

class TileSide(*value*)

Bases: IntEnum

Used to index the sides of a tile. This is the indexing done by the Dustforce engine itself.

TOP = 0

BOTTOM = 1

LEFT = 2

RIGHT = 3

class TileEdgeData(*solid: bool = False*, *visible: bool = False*, *caps: Tuple[bool, bool] = (False, False)*, *angles: Tuple[int, int] = (0, 0)*, *filth_sprite_set:* TileSpriteSet *= TileSpriteSet.NONE_0*, *filth_spike: bool = False*, *filth_caps: Tuple[bool, bool] = (False, False)*, *filth_angles: Tuple[int, int] = (0, 0)*)

Bases: object

Data class for data stored on each tile edge. Many attributes are stored as pairs of data to correspond to the two corners of the tile edge. These corners are ordered clockwise around the tile (the tile should be on your right as you traverse from the first to second corner).

solid: bool = False

Should this edge produce collisions

visible: bool = False

Is this edge visible

caps: Tuple[bool, bool] = (False, False)

Wether an edge cap should be drawn for each corner

angles: Tuple[int, int] = (0, 0)

-90 < angle < 90. Ignored if the corresponding cap flag is set.

Type
Edge join angle in degrees, should be in the range #

filth_sprite_set: *TileSpriteSet* = 0

Sprite set of dust or spikes on this edge. Use *TileSpriteSet.NONE_0* to indicate no filth on this edge.

filth_spike: bool = False

If *filth_sprite_set* is not *TileSpriteSet.NONE_0* this flag controls if there is dust or spikes on the edge.

filth_caps: Tuple[bool, bool] = (False, False)

Same as *caps* but for drawing filth (dust/spikes) caps.

filth_angles: Tuple[int, int] = (0, 0)

Same as *angles* but for filth join angles.

class TileShape(*value*)

Bases: IntEnum

Tiles come in four main types; full, half, big, and small. Images of those tiles can be seen below. Alternatively refer to https://github.com/cmann1/PropUtils/blob/master/files/tiles_reference/TileShapes.jpg for an image of all tiles in one place.

FULL = 0

BIG_1 = 1

SMALL_1 = 2

BIG_2 = 3

SMALL_2 = 4

BIG_3 = 5

SMALL_3 = 6

BIG_4 = 7

SMALL_4 = 8

BIG_5 = 9

SMALL_5 = 10

BIG_6 = 11

SMALL_6 = 12

BIG_7 = 13

SMALL_7 = 14

BIG_8 = 15

SMALL_8 = 16

HALF_A = 17

HALF_B = 18

HALF_C = 19

HALF_D = 20

class Tile(*shape: TileShape = TileShape.FULL*, ***, *tile_flags: int = 4*, *sprite_set: TileSpriteSet = TileSpriteSet.TUTORIAL*, *sprite_tile: int = 1*, *sprite_palette: int = 0*, *_tile_data: Optional[bytes] = None*, *_dust_data: Optional[bytes] = None*)

Bases: `object`

Represents a single tile in a Dustforce level. Positional information (x, y, layer) is stored within the containing `Level` and not in the Tile object itself.

Tiles support the equality and hashing interface.

The constructor will by default create a square virtual tile with 4 zeroed (non-solid nor visible) edges.

shape

The shape of this particular tile.

Type
TileShape

tile_flags

Raw bitmask of flags from the Dustforce engine. In practice this always seems to be 0x4 (which is set by default) corresponding to just the "solid" flag set. In most cases you should just ignore this flag. From the engine the definitions are:

- Bit 1 - "solid slope flag" (probably ignored)
- Bit 2 - "visible flag" (probably ignored)
- Bit 3 - solid flag

Type
3-bit int

edge_data

List[TileEdgeData]: Edge data for each edge of the tile. This should always be a list of length 4 regardless of the tile *shape*.

sprite_set

The sprite set this tile comes from. (e.g. forest, mansion)

Type
TileSpriteSet

sprite_tile

The index of the specific tile within this sprite set (e.g. grass, dirt). Check https://github.com/cmann1/PropUtils/tree/master/files/tiles_reference for a visual reference to get sprite index information.

Type
int

sprite_palette

The color variant of this tile.

Type
int

get_sprite_tuple() → Tuple[*TileSpriteSet*, int, int]

Convenience method for getting a tuple that describes the sprite of a tile for easy sprite comparison and copying.

Returns
A three-tuple containing the sprite set, tile, and palette of this tile.

set_sprite_tuple(*sprite_tuple: Tuple[*TileSpriteSet*, int, int]*) → None

Convenience method for setting sprite information in the same format as *get_sprite_tuple()*.

Parameters
sprite_tuple (*TileSpriteSet*, *int*, *int*) – Sprite set, tile, and palette information.

property sprite_path: str

Gives the path within the extracted sprite metadata to the tile sprite currently selected. You may retrieve the complete game sprites listing from https://www.dropbox.com/s/jm37ew9p74olgca/sprites.zip?dl=0

has_filth() → bool

Returns true if there is filth attached to any edges of this tile

is_dustblock() → bool

Returns true if the tile is a dustblock tile. This is calculated based on the current sprite information.

set_dustblock() → None

Update *sprite_tile* and *sprite_palette* to be the dustblock matching the current *sprite_set*.

Raises
ValueError – If there is no dustblock tile for the current sprite set.

transform(*mat: TxMatrix*) → None

Performs a flip and rotation as dictated by the tranformation matrix. Does not do any kind of scaling or skewing beyond that. Use *upscale()* if you want to increase tile scale.

mat

The transformation matrix. See *dustmaker.level.Level.transform()*

upscale(*factor: int*) → Generator[Tuple[int, int, *Tile*], None, None]

Upscales a tile, returning a list of (dx, dy, tile) tuples giving the relative position of the upscaled tiles and the new tile shape. This is primarily used by *dustmaker.level.Level.upscale()*.

Yields
A three tuple (dx, dy, ntile) where (dx, dy) are the relative position within the scaled up *factor* x *factor* tile square formed and *ntile* is the tile that belongs at that position.

SPRITE_SET_DUSTBLOCK_TILE

Mapping of *TileSpriteSet* to the corresponding dustblock index for that sprite set. Gives -1 if no dustblock tile is available for the given sprite set.

Type:
tuple mapping *TileSpriteSet* -> int

SHAPE_ORDERED_SIDES

A mapping of *TileShape* to a sequence of sides.

For: *TileShape.FULL* this is ordered clockwise starting with the top side.

For half tiles and small slants the ordering starts on the diagonal edge and procedes clockwise around the tile.

For big slants the ordering starts on the diagonal, then the opposite side, then the flat side that's not present on the small slants. For BIG_1 through BIG_4 this is clockwise, for BIG_5 through BIG_8 this is counter-clockwise.

Type:
tuple mapping *TileShape* -> (TileSide, ...)

SHAPE_VERTEXES

Mapping of *TileShape* to the vertex coordinates of the tile in half-tile units. Vertexes are listed top-left, top-right, bottom-right, and bottom-left order.

Type:
tuple mapping *TileShape* -> ((int, int), (int, int), (int, int), (int, int))

SIDE_CLOCKWISE_INDEX

Mapping of *TileSide* to the index of that tile side when sides are listed in clockwise order. This is useful for computing the edge vertexes for a given side from *SHAPE_VERTEXES*.

Type:
tuple mapping *TileSide* -> int

Examples

```
shape, side = TileShape.BIG_1, TileSide.TOP
ind = SIDE_CLOCKWISE_INDEX[side]
vert_a = SHAPE_VERTEXES[shape][ind]
vert_b = SHAPE_VERTEXES[shape][(ind + 1) % 4]
# vert_a = (0, 0), vert_b = (2, 1)
```

CHAPTER

FOUR

VARIABLE MODULE

Module defining the Dustforce variable representation

class VariableType(*value*)

Bases: `IntEnum`

Enumeration of Var type IDs.

NULL = 0

Special code used in the internal format. Null variables cannot be created.

BOOL = 1

INT = 2

UINT = 3

FLOAT = 4

STRING = 5

Dustforce strings are actually byte arrays

VEC2 = 10

Vec2 is a (float, float) tuple

STRUCT = 14

Generic mapping/object type

ARRAY = 15

class Variable(*value: Any*)

Bases: `object`

Variable base class. Variables are a mechanism Dustforce uses to make structure metadata easily available to the game script. Variables will raise an *AssertionError* if they are created with an invalid *value* attribute.

Variables support the equality and hashing interface.

value

The internal value of this variable. Its type will depend on the actual variable type.

assert_types() → None

Checks if the type of *value* matches our concrete variable type.

Raises

ValueError – *value*'s type is invalid

class VariableBool(*value: bool = False*)

Bases: *Variable*

Represents a boolean variable of type `bool`.

class VariableInt(*value: int = 0*)

Bases: *Variable*

Represents a 32-bit signed int variable of type `int`.

class VariableUInt(*value: int = 0*)

Bases: *Variable*

Represents a 32-bit unsigned int variable of type `int`.

class VariableFloat(*value: float = 0.0*)

Bases: *Variable*

Represents a floating point variable of type `float`.

class VariableString(*value: bytes = b''*)

Bases: *Variable*

Represents a string variable of type `bytes`.

class VariableVec2(*value: Tuple[float, float] = (0.0, 0.0)*)

Bases: *Variable*

Represents a 2-dimensional vector of type *(float, float)*.

class VariableStruct(*value: Optional[Dict[str,* Variable*]] = None*)

Bases: *Variable*

Represents a struct (dictionary) mapping of type *dict[str, Variable]*.

class VariableArray(*element_type: Type[*Variable*], values: Optional[List[*Variable*]] = None*)

Bases: *Variable*, `MutableSequence`

Represents an array variable. Arrays are stored a bit differently because they must explicitly encode the type of their sub-elements (heterogenous arrays are not allowed).

VariableArray implements the `collections.abc.MutableSequence` interface, automatically boxing and unboxing accessed elements.

Parameters

- **element_type** (*type[Variable]*) – Variable type of all elements
- **values** (*list[element_type]*) – Array of variables of type *element_type*

value

A tuple containing the element type and the element list. Prefer using *element_type* and the `MutableSequence` interface provided by VariableArray instead of accessing these elements through *value*.

Type
element_type, list[element_type]

property element_type: Type[*Variable*]

Element type of this array

CHAPTER

FIVE

ENTITY MODULE

Module defining basic entity representations as well as a custom entity object for each entity in Dustforce.

class Entity(*variables: Optional[Dict[str,* Variable*]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: `object`

Base class representing an entity object in a map. Commonly used entities have subclasses of *Entity* to represent them. Those that do not will simply be of type *Entity* and have their *etype* field set to control how the Dustforce engine will interpret the entity.

To add new Entity types simply extend this class and include a *TYPE_IDENTIFIER* class attribute that matches the type identifier used internally by Dustforce. The level reader and writer will then automatically use these types (as long as the classes have been initialized). Additionally you can override the *remap_ids()* and *transform()* methods to handle any special processing this entity type requires. If you make these changes consider contributing your entity specialization as a pull request.

etype

The entity type name. This typically matches the *TYPE_IDENTIFIER* attribute of the concrete type of this object.

Type
str

variables

Persist data variable mapping for this entity

Type
dict[str, *Variable*]

rotation

Clockwise rotation of the entity ranging from 0 to 0xFFFF. 0x4000 corresponds to a 90 degree rotation, 0x8000 to 180 degrees, 0xC000 to 270 degrees. This rotation is logically applied after any flips have been applied.

Type
16-bit uint

layer

Layer to render the entity in

Type
8-bit uint

flip_x

Flip the entity horizontally

Type
bool

flip_y

Flip the entity vertically

Type
bool

visible

Is the entity visible

Type
bool

remap_ids(*id_map: Dict[int, int]*) → None

Overridable method to allow an entity to remap any internally stored IDs.

transform(*mat: TxMatrix*) → None

Generic transform implementation for entities. Transforms the Entity *rotation* and *flip_y* attributes (*flip_x* is redundant).

Many subtypes will perform additional transformations on their *variables*.

class Emitter(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Emitter entity class

TYPE_IDENTIFIER = 'entity_emitter'

property e_rotation

Wrapper around *variables['e_rotation']* of type `VariableInt`. Defaults to *0*.

Type
int

property draw_depth_sub

Wrapper around *variables['draw_depth_sub']* of type `VariableUInt`. Defaults to *0*.

Type
int

property r_rotation

Wrapper around *variables['r_rotation']* of type `VariableBool`. Defaults to *False*.

Type
bool

property r_area

Wrapper around *variables['r_area']* of type `VariableBool`. Defaults to *False*.

Type
bool

property width

Wrapper around *variables['width']* of type `VariableInt`. Defaults to *480*.

Type
int

property height

Wrapper around *variables['height']* of type `VariableInt`. Defaults to *480*.

Type
int

property emitter_id

Wrapper around *variables['emitter_id']* of type `VariableUInt`. Defaults to *0*.

Type
int

class CheckPoint(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Checkpoint entity class

TYPE_IDENTIFIER = 'check_point'

transform(*mat: TxMatrix*) → None

Transform the trigger area

property trigger_areas

Wrapper around *variables['trigger_area']* of type *VariableArray[VariableVec2]*.

Type
MutableSequence[(float, float)]

class EndZone(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *CheckPoint*

Proximity based end zone (purple flag) entity class

TYPE_IDENTIFIER = 'level_end_prox'

property finished

Wrapper around *variables['finished']* of type `VariableBool`. Defaults to *False*.

Type
bool

class Trigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Trigger entity entity class

TYPE_IDENTIFIER = 'base_trigger'

property width

Wrapper around *variables['width']* of type `VariableInt`. Defaults to *500*.

Type
int

class FogTrigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Fog trigger entity class

TYPE_IDENTIFIER = 'fog_trigger'

normalize() → None

Resizes *gradient*, *colours*, and *pers* arrays to be the correct length for the given *has_sub_layers* setting.

static get_layer_index(*layer: int*, *sublayer: Optional[int] = None*) → int

Helper function to find fog data for a given layer/sublayer.

Parameters

- **layer** (*int*) – Must be between 0 and 20 inclusive.
- **sublayer** (*int*) – Must be between 0 and 24 inclusive.

Returns

Index into *colours* and *pers* where fog data is stored for the given *layer* and (if present) *sublayer*.

property speed

Wrapper around *variables['fog_speed']* of type `VariableFloat`. Defaults to *5.0*.

Type

float

property gradient

Wrapper around *variables['gradient']* of type *VariableArray[VariableUInt]*.

Type

MutableSequence[int]

property gradient_middle

Wrapper around *variables['gradient_middle']* of type `VariableFloat`. Defaults to *0.0*.

Type

float

property star_bottom

Wrapper around *variables['star_bottom']* of type `VariableFloat`. Defaults to *0.0*.

Type

float

property star_middle

Wrapper around *variables['star_middle']* of type `VariableFloat`. Defaults to *0.4*.

Type

float

property star_top

Wrapper around *variables['star_top']* of type `VariableFloat`. Defaults to *1.0*.

Type

float

property has_sub_layers

Controls if sublayer fog data is enabled for this trigger Defaults to *False*.

Type
bool

property colours

Fog colour in 0xRRGGBB format for each (sub)layer.

Type
MutableSequence[int]

property pers

Mixing coefficient for the fog each (sub)layer from 0.0 to 1.0.

Type
MutableSequence[float]

class AmbienceTrigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Ambience trigger entity class

TYPE_IDENTIFIER = 'ambience_trigger'

property speed

Wrapper around *variables['ambience_speed']* of type `VariableFloat`. Defaults to *5*.

Type
float

property sound_names

Wrapper around *variables['sound_ambience_names']* of type *VariableArray[VariableString]*.

Type
MutableSequence[bytes]

property sound_vols

Wrapper around *variables['sound_ambience_vol']* of type *VariableArray[VariableFloat]*.

Type
MutableSequence[float]

class MusicTrigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Music trigger entity class

TYPE_IDENTIFIER = 'music_trigger'

property speed

Wrapper around *variables['music_speed']* of type `VariableFloat`. Defaults to *5*.

Type
float

property sound_names

Wrapper around *variables['sound_music_names']* of type *VariableArray[VariableString]*.

Type
MutableSequence[bytes]

property sound_vols

Wrapper around *variables['sound_music_vol']* of type *VariableArray[VariableFloat]*.

Type
MutableSequence[float]

class SpecialTrigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Max special trigger entity class

TYPE_IDENTIFIER = 'special_trigger'

class TextTrigger(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Text trigger entity class

TYPE_IDENTIFIER = 'text_trigger'

property hide

Wrapper around *variables['hide']* of type `VariableBool`. Defaults to *False*.

Type
bool

property text

Wrapper around *variables['text_string']* of type `VariableString`. Defaults to *b''*.

Type
bytes

class DeathZone(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Death zone entity class

TYPE_IDENTIFIER = 'kill_box'

transform(*mat: TxMatrix*) → None

Transform the death zone width and height

property width

Wrapper around *variables['width']* of type `VariableInt`. Defaults to *0*.

Type
int

property height

Wrapper around *variables['height']* of type `VariableInt`. Defaults to *0*.

Type
int

class AIController(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

AI controller node entity class

TYPE_IDENTIFIER = 'AI_controller'

remap_ids(*id_map: Dict[int, int]*) → None

Remap the puppet id.

transform(*mat: TxMatrix*) → None

Transform the controller waypoints.

property nodes

Wrapper around *variables['nodes']* of type *VariableArray[VariableVec2]*.

Type
MutableSequence[(float, float)]

property node_wait_times

Wrapper around *variables['nodes_wait_time']* of type *VariableArray[VariableInt]*.

Type
MutableSequence[int]

property puppet

Wrapper around *variables['puppet_id']* of type `VariableUInt`. Defaults to *0*.

Type
int

class CameraNodeType(*value*)

Bases: `IntEnum`

Enum defining the different camera node types

NORMAL = 1

DETACH = 2

CONNECT = 3

INTEREST = 4

FORCE_CONNECT = 5

class CameraNode(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Camera node entity class

TYPE_IDENTIFIER = 'camera_node'

remap_ids(*id_map: Dict[int, int]*) → None

Remap the connected camera node IDs.

transform(*mat: TxMatrix*) → None

Transform the camera zoom and width

property node_type

Camera node type, see *CameraNodeType* enum. Defaults to *1*.

Type
int

property test_widths

Wrapper around *variables['test_widths']* of type *VariableArray[VariableInt]*.

Type
MutableSequence[int]

property nodes

Wrapper around *variables['c_node_ids']* of type *VariableArray[VariableUInt]*.

Type
MutableSequence[int]

property control_widths

Wrapper around *variables['control_widths']* of type *VariableArray[VariableVec2]*.

Type
MutableSequence[(float, float)]

property zoom

Wrapper around *variables['zoom_h']* of type `VariableInt`. Defaults to *1080*.

Type
int

property width

Wrapper around *variables['width']* of type `VariableInt`. Defaults to *520*.

Type
int

class LevelEnd(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Level end flag class

TYPE_IDENTIFIER = 'level_end'

remap_ids(*id_map: Dict[int, int]*) → None

Remap entity IDs attached to this flag.

property entities

Wrapper around *variables['ent_list']* of type *VariableArray[VariableUInt]*.

Type
MutableSequence[int]

property finished

Wrapper around *variables['finished']* of type `VariableBool`. Defaults to *False*.

Type
bool

class ScoreBook(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Score book class

TYPE_IDENTIFIER = 'score_book'

property book_type

Wrapper around *variables['book_type']* of type `VariableString`. Defaults to *b''*.

Type
bytes

class LevelDoor(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Level door class

TYPE_IDENTIFIER = 'level_door'

property file_name

Wrapper around *variables['file_name']* of type `VariableString`. Defaults to *b''*.

Type
bytes

property door_set

Wrapper around *variables['door_set']* of type `VariableInt`. Defaults to *0*.

Type
int

class RedKeyDoor(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Red key door class

TYPE_IDENTIFIER = 'giga_gate'

property keys_needed

Wrapper around *variables['key_needed']* of type `VariableInt`. Defaults to *1*.

Type
int

class EntityHittable(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Base class for all 'hittable' types

property scale

Wrapper around *variables['dm_scale']* of type `VariableFloat`. Defaults to *1.0*.

Type
float

transform(*mat: TxMatrix*) → None

Adjust the entity scale

class Enemy(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *EntityHittable*

Base class for all enemy types

Subclasses can override *FILTH* to control how much "filth" is attributed to this entity. The DF file format requires a totalling of all filth in a level for completion calculations.

FILTH: int = 1

class EnemyLightPrism(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Light prism entity class

TYPE_IDENTIFIER = 'enemy_tutorial_square'

class EnemyHeavyPrism(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Heavy prism entity class. Note that although heavy prisms reward 3 dust when cleansing them they only count as 1 filth from the perspective of completion calculations.

TYPE_IDENTIFIER = 'enemy_tutorial_hexagon'

class EnemySlimeBeast(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Slime beast entity class

TYPE_IDENTIFIER = 'enemy_slime_beast'

FILTH: int = 9

class EnemySlimeBarrel(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Slime barrel (paint can) entity class

TYPE_IDENTIFIER = 'enemy_slime_barrel'

FILTH: int = 3

class EnemySpringBall(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Spring ball/blob entity class

TYPE_IDENTIFIER = 'enemy_spring_ball'

FILTH: int = 5

class EnemySlimeBall(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Slime ball (lab turkey) entity class

TYPE_IDENTIFIER = 'enemy_slime_ball'

FILTH: int = 3

class EnemyTrashTire(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Trash tire entity class

TYPE_IDENTIFIER = 'enemy_trash_tire'

FILTH: int = 3

property max_fall_speed

Wrapper around *variables['max_fall_speed']* of type `VariableFloat`. Defaults to *800.0*.

Type
float

class EnemyTrashBeast(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Trash beast (golem) entity class

TYPE_IDENTIFIER = 'enemy_trash_beast'

FILTH: int = 9

class EnemyTrashCan(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Trash can entity class

TYPE_IDENTIFIER = 'enemy_trash_can'

FILTH: int = 9

class EnemyTrashBall(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Trash ball entity class

TYPE_IDENTIFIER = 'enemy_trash_ball'

FILTH: int = 3

class EnemyBear(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Bear entity class

TYPE_IDENTIFIER = 'enemy_bear'

FILTH: int = 9

class EnemyTotemLarge(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Large totem (stoneboss) entity class

TYPE_IDENTIFIER = 'enemy_stoneboss'

FILTH: int = 12

class EnemyTotemSmall(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Totem entity class

TYPE_IDENTIFIER = 'enemy_stonebro'

FILTH: int = 3

class EnemyPorcupine(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Porcupine entity class

TYPE_IDENTIFIER = 'enemy_porcupine'

class EnemyWolf(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Wolf entity class

TYPE_IDENTIFIER = 'enemy_wolf'

FILTH: int = 5

class EnemyTurkey(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Turkey (critter) entity class

TYPE_IDENTIFIER = 'enemy_critter'

FILTH: int = 3

class EnemyFlag(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Flag entity class

TYPE_IDENTIFIER = 'enemy_flag'

FILTH: int = 5

class EnemyScroll(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Scroll entity class

TYPE_IDENTIFIER = 'enemy_scrolls'

class EnemyTreasure(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Treasure entity class

TYPE_IDENTIFIER = 'enemy_treasure'

class EnemyChestTreasure(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Chest that spawns treasures entity class

TYPE_IDENTIFIER = 'enemy_chest_treasure'

FILTH: int = 9

class EnemyChestScrolls(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Chest that spawns scross entity class

TYPE_IDENTIFIER = 'enemy_chest_scrolls'

FILTH: int = 9

class EnemyButler(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Butler entity class

TYPE_IDENTIFIER = 'enemy_butler'

class EnemyMaid(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Maid entity class

TYPE_IDENTIFIER = 'enemy_maid'

class EnemyKnight(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Knign entity class

TYPE_IDENTIFIER = 'enemy_knight'

FILTH: int = 9

class EnemyGargoyleBig(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Big (punching) gargoyle entity class

TYPE_IDENTIFIER = 'enemy_gargoyle_big'

FILTH: int = 5

class EnemyGargoyleSmall(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Gargoyle (mansion turkey) entity class

TYPE_IDENTIFIER = 'enemy_gargoyle_small'

FILTH: int = 3

class EnemyBook(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Book entity class

TYPE_IDENTIFIER = 'enemy_book'

FILTH: int = 3

class EnemyHawk(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Hawk entity class

TYPE_IDENTIFIER = 'enemy_hawk'

FILTH: int = 3

class EnemyKey(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Key entity class

TYPE_IDENTIFIER = 'enemy_key'

FILTH: int = 1

remap_ids(*id_map: Dict[int, int]*) → None

Remap the door ID.

transform(*mat: TxMatrix*) → None

Transform the last know coordinates.

property door

ID of door entity Defaults to *0*.

Type
int

property lastX

Wrapper around *variables['lastKnowX']* of type `VariableFloat`. Defaults to *0.0*.

Type
float

property lastY

Wrapper around *variables['lastKnowY']* of type `VariableFloat`. Defaults to *0.0*.

Type
float

class EnemyDoor(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Enemy*

Door entity class

TYPE_IDENTIFIER = 'enemy_door'

FILTH: int = 0

class Apple(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *EntityHittable*

Apple entity class

TYPE_IDENTIFIER = 'hittable_apple'

class DustCharacter(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *EntityHittable*

Normal playable dust character entity types

class Dustman(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Dustman entity class

TYPE_IDENTIFIER = 'dust_man'

class Dustgirl(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Dustgirl entity classs

TYPE_IDENTIFIER = 'dust_girl'

class Dustkid(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Dustkid entity class

TYPE_IDENTIFIER = 'dust_kid'

class Dustworth(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Dustworth entity class

TYPE_IDENTIFIER = 'dust_worth'

class Dustwraith(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Dustwraith entity class

TYPE_IDENTIFIER = 'dust_wraith'

class Leafsprite(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Leaf sprite entity class

TYPE_IDENTIFIER = 'leaf_sprite'

class Trashking(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Trash king entity class

TYPE_IDENTIFIER = 'trash_king'

class Slimeboss(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *DustCharacter*

Slime boss entity class

TYPE_IDENTIFIER = 'slime_boss'

class CustomScoreBook(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Entity*

Custom score book (tome) entity class

TYPE_IDENTIFIER = 'custom_score_book'

property level_list

ID of StringList entity Defaults to *0*.

Type
int

class StringList(*variables: Optional[Dict[str, Variable]] = None*, *rotation=0*, *layer=18*, *flip_x=False*, *flip_y=False*, *visible=True*)

Bases: *Trigger*

Data container entity

TYPE_IDENTIFIER = 'z_string_list'

property data

Wrapper around *variables['list']* of type *VariableArray[VariableString]*.

Type
MutableSequence[bytes]

CHAPTER

SIX

PROP MODULE

Module containing dustmaker's prop representation.

class Prop(*layer_sub: int*, *rotation: int*, *flip_x: bool*, *flip_y: bool*, *scale: float*, *prop_set: int*, *prop_group: int*, *prop_index: int*, *palette: int*)

Bases: `object`

Class respresenting a static prop in a map.

To find appropriate values for *prop_set*, *prop_group*, and *prop_index* check out https://github.com/cmann1/PropUtils/tree/master/files/prop_reference.

layer_sub

The sublayer the prop is rendered on. Note that the prop layer is actually stored within the containing *dustmaker.level.Level* itself.

Type
int

rotation

Clockwise rotation of the prop ranging from 0 to 0xFFFF. 0x4000 corresponds to a 90 degree rotation, 0x8000 to 180 degrees, 0xC000 to 270 degrees. This rotation is logically applied after any flips have been applied.

Type
16-bit uint

flip_x

Flip the prop horizontally

Type
bool

flip_y

Flip the prop vertically

Type
bool

scale

Prop scaling factor. This is only available in *LevelType.DUSTMOD* type maps and is fairly coarse in the resolution of the scaling factor.

Type
float

prop_set

Identifier indicating what prop set this prop comes from. This appears to match *dustmaker.tile.TileSpriteSet*.

Type
int

prop_group

Identifier indicating what prop group this prop comes from.

Type
int

prop_index

Index of the desired prop sprite.

Type
int

palette

The colour variant of the prop to render.

Type
int

transform(*mat: TxMatrix*) → None

Performs the requested transformation on the prop's *rotation* and *flip_y* attributes.

CHAPTER

SEVEN

REPLAY MODULE

Module defining the replay container types

LATEST_VERSION = 4

Latest replay version supported by dustmaker/dustmod

class IntentStream(*value*)

Bases: IntEnum

Enumeration of the different intent streams in the order they are listed within the replay binary format.

X = 0

-1 for left, 0 for neutral, 1 for right

Type
X intent

Y = 1

-1 for up, 0 for nuetral, 1 for down

Type
Y intent

JUMP = 2

0 for not pressed, 1 for pressed and unused, 2 for pressed and used.

Type
jump intent

DASH = 3

0 for not pressed, 1 for pressed and unused, 2 for pressed and used (2 is only present for weird subframe things).

Type
dash intent

FALL = 4

0 for not pressed, 1 for pressed and unused, 2 for pressed and used (2 is only present for weird subframe things).

Type
fall intent

LIGHT = 5

0 for not pressed, 10 for pressed and unused, 11 for pressed and used, 1-9 counts down from 10 after the key is released and unused, during this time the intent will be consumed if possible from the player state.

Type
light intent

HEAVY = 6

0 for not pressed, 10 for pressed and unused, 11 for pressed and used, 1-9 counts down from 10 after the key is released and unused, during this time the intent will be consumed if possible from the player state.

Type
heavy intent

TAUNT = 7

0 for not pressed, 1 for pressed and unused, 2 for pressed and used.

Type
taunt intent

MOUSE_X = 8

float in the range [0.0, 1.0] where 0.0 corresponds to the left of the screen and 1.0 corresponds to the right of the screen. This is internally stored with 16 bits of accuracy.

Type
mouse x

MOUSE_Y = 9

float in the range [0.0, 1.0] where 0.0 corresponds to the top of the screen and 1.0 corresponds to the bottom of the screen. This is internally stored with 16 bits of accuracy.

Type
mouse y

MOUSE_STATE = 10

bit mask of current mouse state as defined in *MouseState*.

Type
mouse state

class MouseState(*value*)

Bases: IntFlag

Mouse state bitmask values associated with the *IntentStream.MOUSE_STATE* intent.

WHEEL_UP = 1

WHEEL_DOWN = 2

LEFT_CLICK = 4

RIGHT_CLICK = 8

MIDDLE_CLICK = 16

class Character(*value*)

Bases: IntEnum

Numeric character IDs for each playable character

DUSTMAN = 0

DUSTGIRL = 1

DUSTKID = 2

DUSTWORTH = 3

SLIMEBOSS = 4

TRASHKING = 5

LEAFSPRITE = 6

DUSTWRAITH = 7

class PlayerData(*character: ~dustmaker.replay.Character = Character.DUSTMAN*, *intents: ~typing.Dict[~dustmaker.replay.IntentStream, ~typing.List[~typing.Any]] = <factory>*)

Bases: `object`

Container class for a single player's replay data

character: *Character* = 0

intents: Dict[*IntentStream*, List[Any]]

The intent data parsed from the replay file. These may have length smaller than the number of frames in the replay in which case the neutral value for that intent should be considered the active intent on those frames. Use *:meth: `get_intent_value* to automatically deal with this when reading replays.

get_intent_value(*intent:* IntentStream, *frame: int*) → Any

Returns the value for the given intent at the given frame

class EntityFrame(*frame: int = 0*, *x_pos: float = 0.0*, *y_pos: float = 0.0*, *x_speed: float = 0.0*, *y_speed: float = 0.0*)

Bases: `object`

Container class for a single frame worth of entity desync data.

frame: int = 0

Frame timer for this entity frame.

x_pos: float = 0.0

X position of the entity. This has a resolution of a tenth of a pixel.

y_pos: float = 0.0

Y position of the entity. This has a resolution of a tenth of a pixel.

x_speed: float = 0.0

X-speed of the entity. This has a resolution of 0.01 pixels/s.

y_speed: float = 0.0

Y-speed of the entity. This has a resolution of 0.01 pixels/s.

class EntityData(*frames: ~typing.List[~dustmaker.replay.EntityFrame] = <factory>*)

Bases: `object`

Container class for all the desync frame data for an entity. Note that the engine only stores desync data every 8 frames (and then slower than that eventually for long replays) and only stores the data if the entity moved significantly.

frames: List[*EntityFrame*]

Frame data for the given entity. This should appear in increasing order of frame time but the times might not increase at the same rate.

class Replay(*version: int = 4*, *username: bytes = b''*, *level: bytes = b''*, *frames: int = 0*, *players: ~typing.List[~dustmaker.replay.PlayerData] = <factory>*, *entities: ~typing.Dict[int, ~dustmaker.replay.EntityData] = <factory>*)

Bases: `object`

Container class for a replay

version: int = 4

The format of the replay file. If writing a replay just use LATEST_VERSION unless you want compatibility with vanilla.

username: bytes = b''

The username associated with the replay. If this is unset no username header will be included. If feeding the replay binary directly to dustmod make sure to include a username as it does expect to find a replay header.

level: bytes = b''

The level filename associated with the replay.

frames: int = 0

The length of the replay in frames.

players: List[*PlayerData*]

Per-player replay data

entities: Dict[int, *EntityData*]

Entity desync data per entity. This maps the entity ID in the map to EntityData captured for that entity. Camera entities get the special ID of 1 + 2 * player_index and player entities get the special ID of 2 * player_index (alternatively you can use *get_camera_entity_data()* *get_player_entity_data()* to access these data).

get_player_entity_data(*player: int = 1*) → Optional[*EntityData*]

Get the entity data for the given player entity.

Parameters

player (*int, optional*) – The player to get the entity data for indexed from 1

get_camera_entity_data(*player: int = 1*) → Optional[*EntityData*]

Get the entity data for the camera following the given player.

Parameters

player (*int, optional*) – The player to get the camera of indexed from 1

CHAPTER

EIGHT

BITIO MODULE

Module defining the core binary reader for Dustforce binary formats.

class BitIO(*data: BinaryIO*, *, *noclose: bool = False*)

Bases: `object`

Wrapper around a binary IO source that allows integers to be read/written to. Reads and writes of integers are all serialized to bits in little endian order.

Within the IO source bits are ordered from LSB to MSB. Therefore the first bit of a stream is the '1's place of the first byte. The last bit of a stream is the '128's place bit of the last byte.

Parameters
noclose (*bool*) – Normally when the BitIO object is closed *data* is also closed. If this is set then *data* will be left open after this BitIO is closed.

data

A binary data stream. Should support read/write/seek if those respective operations are done on the BitIO object itself.

Type
BinaryIO

release() → None

Prevents *close()* from closing *data* as well.

close() → None

Flush any pending bits and close *data* unless it has been released.

aligned() → bool

Returns True if the stream is aligned at a byte boundary.

align() → None

Seeks the stream forward to the nearest byte boundary. This does not require *data* to support seek itself.

skip(*bits: int*) → None

Skips *bits* bits in the bit stream. Requires *data* to support seeks.

Parameters
bits (*int*) – the number of bits to skip

bit_tell() → int

Returns the current bit position of the stream

bit_seek(*pos: int*) → None

Seeks to a new bit-position in the stream.

Parameters
pos (*int*) – The position in bits from the start of the stream

class BitIOReader(*data: BinaryIO*, *, *noclose: bool = False*)

Bases: *BitIO*

Bit reader wrapper for a data stream

read(*bits: int*, *signed: bool = False*) → int

Reads in the next *bits* bits into an integer in little endian order.

Parameters
- **bits** (*int*) – The number of bits to read in
- **signed** (*bool*) – Wether the most significant bit should be interpretted as a sign bit.

read_bytes(*num: int*) → bytes

Reads in the next *num* bytes and returns them as a *bytes* object.

Parameters
num (*int*) – The number of bytes to extract from the bit stream.

data

A binary data stream. Should support read/write/seek if those respective operations are done on the BitIO object itself.

Type
BinaryIO

class BitIOWriter(*data: BinaryIO*, *, *noclose: bool = False*)

Bases: *BitIO*

Bit writer wrapper for a data stream.

write(*bits: int*, *val: int*) → None

Writes *val*, an integer of *bits* bits in size, to the stream.

Note: If the last byte is partially completed it will not be written until the stream is closed or flushed.

write_bytes(*buf: bytes*) → None

Writes the bytes in *buf* to the stream

Parameters
buf (*bytes*) – The data to write to the stream

close() → None

Flush any pending bits and close the underlying stream (unless released).

flush() → None

Flushes any trailing bits.

Warning: If there are trailing bits this will cause the stream to seek forward to the next byte boundary. Generally you shouldn't need to call this directly and should allow other methods like *close()*, *align()*, *bit_seek()* to call flush for you at times that always make sense.

align() → None

Seeks the stream forward to the nearest byte boundary. This does not require *data* to support seek itself. This also triggers a flush.

data

A binary data stream. Should support read/write/seek if those respective operations are done on the BitIO object itself.

Type
BinaryIO

bit_seek(*pos: int*, *, *allow_unaligned: bool = False*) → None

Seeks to the desired position in the stream relative the start. This also triggers a flush of any pending data at our current location.

Parameters

- **pos** (*int*) – The bit position to seek to relative the start of the stream.
- **allow_unaligned** (*bool*) – Normally unaligned seeks are not allowed. If you set this flag they will be allowed however any write performed at the new location will have the effect of zero'ing any bits earlier within the byte that we are seeking into.

Warning: Seeking into a non-byte aligned position is not well supported and cannot be done generally without performing a read.

Raises
RuntimeError – If seek is not byte aligned and *allow_unaligned* is not set.

CHAPTER

NINE

DFREADER MODULE

Module providing methods for reading Dustforce binary formats including level files.

class DFReader(*data: BinaryIO*, *, *noclose: bool = False*)

Bases: *BitIOReader*

Helper class to read Dustforce binary files

read_expect(*data: bytes*) → None

Ensure the next bytes match *data*

Raises
LevelParseException – If the read bytes do not match *data*.

read_float(*ibits: int*, *fbits: int*) → float

Read a float in the Dustforce format

Parameters

- **ibits** (*int*) – Number of integer bits
- **fbits** (*int*) – Number of fractional bits

read_6bit_str() → str

Read a '6-bit' string. These are strings with length between 0 and 63 inclusive that contain only alphanumeric lower and uppercase characters in addition to '_' and '{'.

read_variable(*vtype:* VariableType) → *Variable*

Read a variable of a given type.

Parameters
vtype (VariableType) – The type of variable to read

read_variable_map() → Dict[str, *Variable*]

Convenience method equivalent to *read_variable(VariableType.STRUCT).value*

read_segment(*level:* Level, *xoffset: int*, *yoffset: int*) → None

Read segment data into the passed level. In most cases you should just use *read_level()* instead of this method.

Parameters

- **level** (Level) – The level object to read data into
- **xoffset** (*int*) – The segment x-offset in tiles
- **yoffset** (*int*) – The segment y-offset in tiles

read_region(*level:* Level) → None

Read region data into the passed level. In most cases you should just use *read_level()* instead of this method.

Parameters
level (Level) – The level object to read data into

read_var_file(*header: bytes*) → Dict[str, *Variable*]

Reads a variable mapping with a given header. There are several file types that Dustforce use that are expressed this way including notably "stats1" (header=b"DF_STA") and "config" (header=b"DF_CFG").

Parameters
header (*bytes*) – The expected file header at the start of the stream. Just pass b"" if you've already read and checked the header.

read_level_ex() → Tuple[*Level*, List[int]]

Extended version of *read_level()*.

Read level file metadata into a Level object while extracting additional metadata so that the rest of the data can be ingested in an opaque way. *read_level_ex* ends with the reader byte-aligned. The entirety of the region data can be read subsequently with reader.read_bytes(region_bytes) or using *read_region()*.

This can be used with *dustmaker.dfwriter.DFWriter.write_level_ex()* to modify level metadata without reading in region data.

Example:

```
# Re-write level metadata without reading in region data.
level, region_offsets = reader.read_level_ex()
region_data = reader.read_bytes(region_offsets[-1])
...
writer.write_level_ex(level, region_offsets, region_data)
```

Example:

```
# Manually read region data
level, region_offsets = reader.read_level_ex()
for _ in region_offsets[:-1]:
    reader.read_region(level)
```

Returns

(level, region_offsets) tuple

- **level**
  dustmaker.level.Level object with metadata (e.g. level.variables and level.sshot) filled in.
- **region_offsets**
  list of byte offsets of each region from the current stream position (which is aligned). The last element of this array is the end of the region data and does not correspond to a region itself.

read_level(*, *metadata_only: bool = False*) → *Level*

Read a level data stream and return the *dustmaker.level.Level* object.

Parameters
metadata_only (*bool, optional*) – If set to True only the variables and sshot data will be set in the returned Level.

Raises
LevelParseException – Parser ran into unexpected data.

read_replay(*, *known_length: Optional[int] = None*) → *Replay*

Read in a replay from the input stream.

Parameters
known_length (*int, optional*) – The total length in bytes of the replay. Giving this length beforehand can speed up parsing the replay data.

data

A binary data stream. Should support read/write/seek if those respective operations are done on the BitIO object itself.

Type
BinaryIO

read_level(*data: bytes*) → *Level*

Convenience function to read in a level from bytes directly

Parameters
data (*bytes*) – The data source for the level

Returns
The parsed Level object.

CHAPTER

TEN

DFWRITER MODULE

Module providing methods for write Dustforce binary formats including level files.

class DFWriter(*data: BinaryIO*, ***, *noclose: bool = False*)

Bases: *BitIOWriter*

Helper class to write Dustforce binary files

write_float(*ibits: int*, *fbits: int*, *val: float*) → None

Write a float *val* to the output stream

Parameters

- **ibits** (*int*) – Number of integer bits
- **fbits** (*int*) – Number of fractional bits
- **val** (*float*) – The floating point number to write

write_6bit_str(*text: str*) → None

Write a '6-bit' string. These are strings with length between 0 and 63 inclusive that contain only alphanumeric lower and uppercase characters in addition to '_' and '{'.

Raises

- **ValueError** – If length of string exceeds 63
- **ValueError** – If invalid character is present

write_variable(*var:* Variable) → None

Write a variable to the output stream. This does not write the type of the variable, that will need to be encoded somewhere else if needed.

Parameters

var (Variable) – The variable to write to the stream.

Raises

- **ValueError** – If a VariableString value is longer than 65535 bytes. Note that structs and arrays each handle their string sub-elements in a way that they may be longer than this limit.
- **ValueError** – If a VariableArray has more than 65535 elements including any continuations if the element type is VariableString.
- ***LevelParseException*** – If *var* is of unknown variable type.

write_var_file(*header: bytes*, *var_data: Dict[str,* Variable*]*) → None

Writes a variable mapping with a given header. There are several file types that Dustforce use that are expressed this way including notably "stats1" (header=b"DF_STA") and "config" (header=b"DF_CFG").

Parameters
header (*bytes*) – The file header at the start of the stream.

write_level_ex(*level: Level*, *region_offsets: List[int]*, *region_data: Union[bytes, Iterable[bytes]]*) → None

Extended version of *write_level()*.

Writes *level* to the output stream. This is the advanced API for efficiently modifiying level metadata. If *region_offsets* is non-empty this will use *region_offsets* and *region_data* to populate the region data section of the output rather than calculating it from *level*.

See *dustmaker.dfreader.DFReader.read_level_ex()* for examples on how to use this in practice.

Parameters

- **level** (Level) – The level file to write
- **region_offsets** (*list[int]*) – Region offset metadata as returned by DFReader.read_level_ex(). If empty this behaves the same as *write_level()*. If you wish to omit all region data pass *[0]* instead.
- **region_data** – (bytes | Iterable[bytes]): Bytes data (or an iterable of bytes data) to write in the region section of the map. If *region_offsets* is empty this argument is ignored.

write_level(*level: Level*) → None

Writes *level* to the output stream. This is equivalent to *write_level_ex(level, [], b"")*.

Parameters
level (Level) – The level file to write

write_replay(*rep: Replay*, *, *force_username=False*) → None

Write *rep* to the output stream.

Parameters

- **rep** (Replay) – The replay object to write.
- **force_username** (*bool, optional*) – Write a username header even if *rep.username* is empty.

data

A binary data stream. Should support read/write/seek if those respective operations are done on the BitIO object itself.

Type
BinaryIO

write_level(*level: Level*) → bytes

Convenience function to write a map file directly to bytes in memory.

Parameters
level (Level) – The level file to write

Returns
The bytes that encode that level file

CHAPTER

ELEVEN

EXCEPTIONS MODULE

Module defining shared exception classes.

exception LevelException

Bases: Exception

Top level dustmaker exception.

exception LevelParseException

Bases: *LevelException*

Exception indicating an error reading a level file.

PYTHON MODULE INDEX

INDEX

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W

X

Y

Z