

Tiled Quick Plugin Usage

Custom properties:

1. “onStart” - (type: *string*) - **JS handler**, called after **loading** scene (applicable for **map, layer, object**);
2. “onExit” - (type: *string*) - **JS handler**, called after **unloading** scene (applicable for **map, layer, object**);
3. “onPress” - (type: *string*) - **JS handler**, called after **press** on object (applicable for **object** which clickable == *true* -> see property 8);
4. “onClick” - (type: *string*) - **JS handler**, called after **click** on object (applicable for **object** which clickable == *true* -> see property 8);
5. “onRelease” - (type: *string*) - **JS handler**, called after **release** object (applicable for **object** which clickable == *true* -> see property 8);
6. “drawDebug” - (type: *bool*) - drawing object by **shape**, painted due **red line** (applicable for **object**);
7. “pressByShape” - (type: *bool*) - *true* - object will be clickable by defined shape, *false* - bounding rectangle (applicable for **object**);
8. “clickable” - (type: *bool*) - *true* - object will can grab touch events: “onPress”, “onClick”, “onRelease” (applicable for **object**);

JS handlers:

JS tutorial: <https://www.w3schools.com/js/default.asp>

JS handlers will be evaluated on loading scene, upon error -> message with reason will be printed to logs.

JS handlers have few predefined variables:

- “object” - instance where attached handler
- “engine” - instance of game engine which contains utilities for control (see *below*)
- “external” - instance of object which controls external events or functional (see *below*)

For print some variable, object, text use:

- *console.log(<content>)*
- *console.info(<content>)*
- *console.warn(<content>)*
- *console.error(<content>)*

For more: <https://doc.qt.io/qt-5/qtquick-debugging.html#console-api>

Tiled item usage in JS:

Tiled item is all objects in tiled scene: *map*, *layer*, *map object*

Some properties and methods you can find by:

- <https://doc.qt.io/qt-5/qquickitem.html#properties>
- <https://doc.qt.io/qt-5/qobject.html#properties>

Also advanced properties:

1. “id” - (type: *int*; *readonly*) - id of object, defined by tiled editor
2. “objectNamePath” - (type: *string*, *readonly*) - full name of object with parents joined via “.” (for example: “layer.sublayer.hero”)

Methods:

1. “getTiledProperty” - (return: *variant*, params: *string*(name)) - get value of property by “name” which added by tiled editor
2. “setTiledProperty” - (return: *void*, params: *string*(name), *variant*(value)) - set value to property by “name” which added by tiled editor

Signals connecting: <https://doc.qt.io/qt-5/qtqml-syntax-signals.html#connecting-signals-to-methods-and-signals>

Connect to signal example:

```
<instance>.<signal>.connect(function() {<handler>})  
animation.finished.connect(function() { console.log(animation, “finished!”) })
```

Map object item usage:

Map object item inheritances all members and methods from Tiled item;

Also advanced properties:

1. “stateMachine” - (type: *StateMachine*; *readonly*) - state machine of object (usage see bellow)
2. “textData” - (type: *TextData*; *readonly*) - text data of object, exists only if object is text (usage see bellow)

Custom tiled editor properties:

1. "drawDebug" - object bounded by red line, "1" view in editor, "2" in application
2. "pressByShape" - clickable areas, "3" - true, "4" - false



State Machine:

Methods:

1. “setState” - (return: *void*, params: *string*(name), *string*(tilesetName), *int*(tileId)), setup state by name to tile from tiles by tilesetName and tileId, tile can be single image or animated sprite which will be looped during state
2. “setInvisibleState” - (return: *void*, params: *string*(name)), setup invisible state by name for object
3. “setTransition” - (return: *void*, params: *string*(from), *string*(to), *string*(tilesetName), *int*(tileId)), setup transition between states, if from/to equal “*” is meaning that state point includes all states

Better location of setup state machine on “onStart” handler.

Example of setup state machine:

```
object.stateMachine.setState("opened", "doors", 10)
object.stateMachine.setState("closed", "doors", 1)
object.stateMachine.setTransition("opened", "closed", "doors-animations", 10)
object.stateMachine.setTransition("closed", "opened", "doors-animations", 1)
object.state = "closed"
```

TextData:

Properties:

1. “text” - (type: *string*) visible text content
2. “font” - (type: *font*) font of text, for more: <https://doc.qt.io/qt-5/qml-font.html>
3. “color” - (type: *color*) color of text, for more: <https://doc.qt.io/qt-5/qml-color.html>
4. “alignment” - (type: *Qt.Alignment*) alignment of text, for more: <https://doc.qt.io/qt-5/qml-qtquick-text.html#horizontalAlignment-prop>
5. “wordWrap” - (type: *bool*) wrap words if text overlap bounding rectangle

Engine:

Methods:

1. “find” - (return: *TiledItem*, params: *int*(id)) find object on scene by id
2. “find” - (return: *TiledItem*, params: *string*(path)) find object on scene by path or full name
3. “getPathAnimation” - (return: *PathAnimation*, params: *TiledItem*(object)) get path animation attached to object
4. “getPathAnimations” - (return: *list<PathAnimation>*, params: *TiledItem*(object)) get path animations attached to object
5. “createPathAnimation” - (return: *PathAnimation*, params: *TiledItem*(object), *MapObjectItem*(path)) create path animation for object by object path, if path is ellipse will be created ellipse *EllipsePathAnimation*, else if path is rectangle, polyline or polygon will be created *PolygonPathAnimation*

PathAnimation:

Animation will be **automatically** destroyed after signal “finished” and reference to animation will be invalid.

Properties:

1. “object” - (type: *TiledItem*, *readonly*) - object which animated
2. “path” - (type: *MapObjectItem*, *readonly*) - path for animation of object
3. “center” - (type: *point*) - center of object for moving by path, default value: (0, 0). (0, 0) -> top, left of object, for create point use: *Qt.point*(<x>, <y>), reference: <https://doc.qt.io/qt-5/qml-point.html>
4. “loops” - (type: *int*) - count of loops playing of animation, default value is 1. If value equals 0 than animation will playing infinitely
5. “backward” - (type: *bool*) - loop will contains two parts, first part is animation in forward direct and second is backward, default value is *false*
6. “running” - (type: *bool*) - state of animation

Methods:

1. “start” - start animation from begin
2. “stop” - stop animation and reset state of animation
3. “reset” - reset animation state, return to begin

Signals:

1. “started” - animations was started
2. “finished” - animation was finished by stop method or end of animation
3. “reached” - animation was finished by end of animation

PolygonPathAnimation:

PolygonPathAnimation inheritances all members of PathAnimation.

Properties:

1. “intervals” - (type: *list<int>*, *initialisation is necessarily before “start”*) intervals in ms for every line of rectangle/polygon/polyline, if specify only one interval (for example: [1000]) then this interval applied for whole path evenly. Count of intervals must be strictly equal to one or the count of lines in shape

EllipsePathAnimation:

EllipsePathAnimation inheritances all members of PathAnimation.

Properties:

1. “interval” - (type: *int*) interval in ms for range
2. “startAngle” - (type: *float*) begin point on ellipse, range: (0-1), default value is 0
3. “endAngle” - (type: *float*) end point on ellipse, range: (0-1), default value is 1

Example of PolygonPathAnimation:

```
var currentAnimation = engine.getPathAnimation(object)
if (currentAnimation) {
    currentAnimation.stop()
}
var path = engine.find("game.heroPath")
var animation = engine.createPathAnimation(object, path)
animation.intervals = [10000]
animation.loops = 1
animation.center = Qt.point(object.width, object.height)
animation.reached.connect(function() {
    object.state = ""
})
animation.start()
```

Example of EllipsePathAnimation:

```
var path = engine.find("game.diamondPath")
var animation = engine.createPathAnimation(object, path)
animation.interval = 1000
animation.endAngle = 0.5
animation.backward = true
animation.center = Qt.point(object.width / 2, object.height / 2)
animation.start()
```

External:

External object has this properties:

1. “scene” - (type: [Scene](#)) - scene object from scenario library
2. “provider” - (type: [QObject](#)) - object which will be provide external functional of game(will be described later)

Scene

Properties:

1. “name” - (type: [string](#), *readonly*) name of scene specified in scenario json
2. “outcome” - (type: [list<string>](#), *readonly*) all available outcomes for scene specified in scenario json
3. “tiledUrl” - (type: [url](#), *readonly*) url of tmx file specified in scenario json
4. “backgroundUrl” - (type: [url](#), *readonly*) url of background image specified in scenario json
5. “provider” - (type: [QObject](#)) - object which will be provide external functional of game(will be described later)

Methods:

- “getStoredVariable” - (return: [variant](#), params: [string](#)(key)) get common variable by key for scenes which will be saved
- “setStoredVariable” - (params: [string](#)(key), [variant](#)(value)) set common variable by key for scenes which will be saved
- “saveStored” - force save all common variables
- “invokeOutcome” - (params: [string](#)(key, default value = “[default](#)”)) - invoke outcome for scene specified in scenario json