How to use the Game Developer API

1 Overview

The Game API is intended to have as little intrusion into the game making process as possible. Games can be made with creation program or by hand, using any assortment of libraries.

The API is provided as a set of Javascript functions that are to be called on specific actions. Exactly how these functions are called is left to the discretion on the implementor, but the game must promise to execute them in given scenarios.

2 The Game API

The API has 7 functions that can be called to make an impact on the user outside of the minigame. The user has backpack which contains items they can use to modify their health, irrespective of any game mechanics the minigame may have. This is NOT to say extra items inside the game cannot also affect the health and status values.

It is upto the game developer to also visualise the backpack; there is no standardised menu to use. The contents of the bag are passed in when the game is launched, and then are updated when an item is used. Beyond this, the backpack can be displayed in any way that is seen fit by the developer.

Additionally, when modifying health or status values, the new value should be read from the callback function. The returned value may not always be exactly what was specified (e.g. if old health was 100 and we take away 20, the value returned may not be 80). For this reason, you should not manually keep track of health. When a users health reaches 0, the game is over and the user is returned to the hub.
The 7 functions are as follows:

finishGame(score, currency)

This should be called when the game ends for any reason. If you wish to display a score screen for the game then this should appear before this function is called. It will unload the game from memory and return to the hub area.
Parameters:

- score: the score of the game
- currency: the currency earned from playing the game

useCarriable(carriableId, cb)

A carriable is an item the user can use to restore health and status values. This function should be called when the user wishes to utilise an item, which are all provided for you in the “bag" object when the game is run.
Parameters:

- carriableId: the id of the carriable to use
- cb: the callback function, explained below

The callback function accepts the parameters (bag, health, statuses, avatarImage, symptoms) which are:

- bag: the updated contents of the players bag
- health: the new health value
- statuses: the new status values
- avatarImage: the new avatar image
- symptoms: the updated list of symptoms

getCarriableInfo(carriableId, cb)

Gets the information for a carriableId, which will be needed to display the information and sprite to the user so they can select it.
Parameters:

- carriableId: the ID of the carriable to query for information
- cb: the callback function, explained below

The callback function accepts the parameters (carriable) which are:

- carriable: the carriable configuration object

modifyHealth(changeVal, cb)

Modifies the health by a **relative** amount. It is important to not the value specified may be modified by the server. It is therefore important to use the value specified in the callback as the new health, and not calculate it yourself.
Parameters:

- changeVal: the amount to add/subtract from the health value
- cb: the callback function, explained below

The callback function accepts the parameters (health, avatar, symptoms) which are:

- health: the new health value
- avatar: the new avatar image
- symptoms: the new list of symptoms

modifyStatus(statusId, changeVal, cb)

Modifies the value of a given status.
Parameters:

- statusId: the ID of the status to affect
- changeVal: the amount to add/subtract from the status value
- cb: the callback function, explained below

The callback function accepts the parameters (id, value) which are:

- id: the id of the specified status (the same as the one supplied)
- value: the new value for the status

getAvatarImage()

Returns the avatar image that is current in use. It is in the form of an image object, not the raw Base64 encoding.

getAssetURL(asset)

This is how extra assets are loaded. Returns the URL of where the asset is located. This must be manipulated into a tag and then added to the canvas. e.g. images must be put in an <img> object, and Javascript files must be put in a <script> tag.
Parameters:

- asset: a string specifying the asset to load relative to the entryObject

3 The Synchronous API

There is a synchronous version of the API for development applications that do not support the asynchronous callback style Javascript offers.

The functions available from this API are the same as already discussed, along with a couple of extra function to facilitate the conversion.

For each function available in the asynchronous API, append “Sync" to the end of the function name to call it’s synchronous counterpart (e.g. “modifyHealthSync"). The parameters you supply are identical, less the callback parameter.

After the function has been called, you must poll the ready state of the function to know when it has returned; otherwise you **WILL** get erroneous data. The function getReady(String) will return the ready state of the specified function. e.g. getReady(“modifyHealthSync"). This will return the integer 0 if the function is not ready, and integer 1 if it is. Note that functions in the asynchrous API that do not take callback functions do not need to be polled as no asynchronous operation occurs (they have synchronous counterparts purely to make it easier to remember).

Once the function has stated it is ready, you can access the data via the getValue(String, String, String). This function takes 1 compulsory String parameter and 2 optional String paramters.

There correspond to the names of the data value you want to get (listed below). e.g getValue(“modifyHealthSync") will return the Javascript object that is set by modifyHealthSync. This object contains a key “health" which contains the users health. Therefore, to retrive just this value we call getValue(“modifyHealthSync", “health"). Simiarly, if the second level is also a Javascript object, the 3 parameter will retrive the key in *that* object. If the supplied keys to not correspond to a value, the Javascript value undefined is returned.

4 How to structure your code

There must be exactly one entry script, which will be the script the framework calls. This entry script can then, in turn, call other scripts and assets as desired. The entry script must contain a function “run" as details below:

function run(api, canvas, assetBaseURL, startHp, statuses, bag)

- a: the Game API, this is the object that will contain the callable functions as detailed earlier
- can : the HTML5 canvas. This is where the display will be rendered.
- assetBaseURL: the base URL for the assets. It is the root folder of the game scripts.
- startHp: the value of health the player currently has
- statuses: all the different statuses the player has (e.g. bloodsugar for diabetes)
- bag: the bag with the players carriable items

5 Setting up the development script

There is a development script available, which will allow you to simulate the server on a local machine. The responses are of course limited, but it will allow you to see the functionality work.

The easiest way to set up local testing is with a few files, and be sure to include all other assets within an “assets" folder:

example.html

The following 4 lines should be written to *example.html*. api.js is the local api testing file, and dummyGame.js is the script containing the *run* function

```
<canvas id="canvas"></canvas>
<script src="api.js"></script>
<script src="dummyGame.js"></script>
<script src="example.js"></script>
```

example.js

This file will set up the canvas environment and launch the game

```
var c = document.getElementById("canvas");
var g = new GameLauncher();
g.launchGame(dummyGame, c);
```

6 Conclusion

With this, you should now be able to create games using the framework, test it locally and be in a ready state for it to be uploaded and used by players!