beAdventurous Engine 1.0

User manual

A vanilla javascript HTML5 engine for making Point & Click adventures with JSON files

Index

Introduction to the engine

The aim of beAdventure Engine is to give users a simple yet powerful way to develop simple Point & Click adventures a la Monkey Island and be able to play them directly from a browser. The engine is written completely in vanilla javascript, no external library needed. Also there is no server access whatsoever so the complete game can be put on modern Cloud storages to gain instant HA and durability with almost 0 costs to maintain the game online.

The main structures are represented by JSON files which describe **maps** and **interactions** between objects and characters in the map. This led to an incredible flexibility both to creating the story and extending the engine itself.

Point & Click Adventures

As many know Point & Click adventures are a type of game where the user interacts with a mouse over different objects and characters in the scene to unlock puzzles and mysteries and usually combines different kinds of actions to exploit the situation and progress with the main story.

Exemples come from history, mainly from LucasArt with titles like Secret of Monkey Island, Maniac Mansion, Days of the Tentacle or Indiana Jones and the Fate of Atlantis.

Point & Click Adventures can be played by anyone as they are not fast paced, and can be really enjoyed at any moment.

What brought me to define an engine is doing something similar to Cable Detective presented by AWS at the 2020 Re:invent. I saw the potential to **engage people not only when they are at a stand but anytime during a convention by bringing them to play a branded game.**

Also, even if this idea was born for work purposes, its nature let you use it for every kind of project, as the possibilities are limitless.

Convention over configuration

The project is developed so the user doesn't need to configure anything in code apart from the JSON files. To do so the Engine **requires the user to follow some convention when creating and saving files** for the game.

Font files

Fonts can be in .otf, .ttf or other **css compatible formats** and must be included in the **font directory** inside the **assets** folder. In order to be recognised by the Engine the user must do 2 things:

1) Add the following line at the top of the **style.css** file inside the **css** folder of the project:
   ```
   @font-face { font-family: <name>;  src:
   url('../assets/fonts/<name>.ttf'); }
   ```
   <name> is very important as it will be referenced by the Engine.
2) Register the font and style inside the engine by doing so:
   ```
   this.fontManager.addFont('<name>', { color: 'white', size: 18 });
   ```
   This is done in **engine.js** in the **start(number)** method line 80. Check the default one as an example.

After doing that, the font can be used in any dialog or interface or static text in the game.

Map files
Map files must be named **map<NUMBER>.json** and must be placed inside the outer **map** folder in the **assets** directory.

Image Files
Image files are in **png format** and one of the core of the game and can cover different roles:

- **Backgrounds**: place them inside **assets/images/maps/map<number>/** calling them as **map<number>_R.png**, **map<number>_M**.png, **map<number>_F.png**, where R is Rear, M is Medium and F is Front. That is because the engine manages 3 backgrounds in 3 different positions to add complexity and flexibility to the scene. Note that these can also be blank 1x1 sprites if you need to do some particular scenes like the intro in the demo.
- **Cursors:** cursors are already predefined inside **assets/images/ui/menu/cursors**, they are in .cur format and can be replaced keeping the same name and created in seconds from this site: https://www.cursor.cc/
- **Interface:** every object in the interface has, for now, a predefined dimension and “hotspot”; if you want to modify the interface just replace the files keeping the name and respecting the proportion of useful areas like dialogs spaces, name spaces or object spaces in the inventory.
- **Objects:** are the static elements in the scene that can have 1 or 2 states depending if the object has the **trigger property** in the JSON configuration or not. Simple objects can be placed in **assets/images/ui/objects** and can be named in any way you want, but keep the name simple and recognizable, possibly in a snake or camel case if needed, as it will be referenced in the JSON file to access the object itself. If the object has 2 states, for example a lever, create 2 files, one for each state, and call them **<name>.png** and **<name>ON.png**. The Engine will automatically use the correct image for the trigger status.
- **Characters:** are the animated element in the screen and can be used cleverly to do more than represent a simple PG on the screen, for example the intro screen is

composed by a character that acts as an animation. They can be **named as you want** but some rules must be respected in order to make them work as expected.
1) Every character .png is a **spritesheet**; the main character spritesheet is defined, for now, as a default spritesheet named **main and has the following structure.**

Please respect the frame count for every animation or alter the **spriteManager.js** related method to reflect your number of frames for each animation. Note, in future i'll hope to parametrize, these values externally to make it easy to configure them.

2) **Other characters can have any number of frames** but must have only one animation in horizontal like the example.

3) Files must be named simply for convenience as they will be referenced in JSON files.

4) Files must be placed in **assets/images/ui/pgs**.

Pro tip: For heavy animations you can save the file in **.gif** or .**jpeg** format, just keep the extension as **.png**, just remember the style of game you are creating, Gifs and Jpegs reduce image quality!

Faces: faces are related to animated characters, they must be **124x124** in .**png** format and have the following structure **<name>Face.png.** They are saved inside the character folder: **assets/images/ui/pgs/**.

Example:

eri.png

eriFace.png

Music files

Music files are in .**mp3** format and must be placed in **assets/musics**. Remember as always to use simple names as they will be referenced in JSON files.

Sound files

Sound files are in .**wav** format and must be placed in **assets/sounds**. Remember as always to use simple names as they will be referenced in JSON files.

Minigames

Minigames lives on their own but must respect 4 important rules:

1) The minigame interface must be copied from **minigame.js** and used as a guide to define the entry and exit point of the minigame
2) The minigame must be placed in its own directory inside **assets/minigames**. Inside that folder you can put your own sounds, musics, images, etc.
3) The minigame style in css, if needed must be placed at the bottom of the **style.css** main file.

4) The minigame js dependencies must be registered after the engine dependencies and the minigame must be registered with a unique name in the engine. Example below:

```
<body>
  <div class="main-game"><canvas id="game"></canvas></div>

  <script src="js/settings.js"></script>
  <script src="js/soundsSystem.js"></script>
  <script src="js/spriteManager.js"></script>
  <script src="js/fontManager.js"></script>
  <script src="js/mouseManager.js"></script>
  <script src="js/interactableManager.js"></script>
  <script src="js/mapManager.js"></script>
  <script src="js/minigame.js"></script>
  <!-- Minigames here -->
  <script src="assets/minigames/fallout-minigame/fallout-minigame.js"></script>
  <!-- ============== -->
  <script src="js/engine.js"></script>

  <script>
    const game = new beAdventureEngine("game");
    game.registerMiniGame("FalloutMinigame", FalloutMinigame);
    game.start(0);
  </script>
</body>
```

Once a minigame is registered in the Engine, it can be called using its name inside a proper interactable.

Engine Structure

The Engine is currently composed of 9 Javascript files each one in charge of managing a specific aspect of a Point & Click game. The purpose of this chapter is to analyse them to give a hint to the developer on how to extend the current version.

Engine.js
This is the main file and calls all other classes to make the game loop cycle work as expected. In this class there are some methods and variables to be aware of.

`gameCanvas;`
This is used to access the game canvas registered in the constructor, with this in all the drawing operations you can access the game context with `.getContext("2d");`

`gameWidth;`
`gameHeight;`
These are used to always get the actual viewport dimensions despite pixel ratio correction

`settings;`
`soundSystem;`
`spriteManager;`
`fontManager;`
`mapManager;`
`mouseManager;`
`interactableManager;`
These are all the other classes included in the main one

`gameStatus;`
`gameVariables;`
These two variables are very important to maintain the status and the global variable coherent across all the engine systems. You can always use them to get access to the most current state of the game and all of its elements obtained from the JSON configuration files.

`constructor(canvasId) {}`
Pass the Game Canvas you've created as an Id to the Engine, that will link the canvas to the engine.

`start(number) {}`
This will be used to start the game passing the first map to use: just set the number e.g: 0 or 1.

`registerMiniGame(name, miniGame) {}`
This will register a minigame to the engine given a unique name to recall it.

`gameLoop(time) {}`
This is the actual game loop and it takes the time given by the `requestAnimationFrame()` method. Inside the method there are 4 important areas:

1) Define the game tick by resolving the internal clock with the cpu time.
2) Manage the game logic, altering all the variables in place.
3) Animate all the game elements, providing updates to the game graphics.
4) Draw the graphics.

`fixDpi() {}`
This method is used to correct aspect ratio on modern screens and scale the canvas context accordingly.

`drawLoader() {}`
`drawMap() {}`
All the draws operations can be controlled by the `gamestatus.levelStatus` variable allowing to separate different draws operations per different game screens. At the moment we have: loader, map and minigame status. Other statuses can be defined as simple numbers and the relative logic and drawing can be put here.

`animateLoader() {}`
`animateMap() {}`
Same as per the drawing operations the animate operations depend on the levelStatus variable.

```
lerp (start, end, amt){}
```
This is a simple Lerp function to help during movement and animation transitions.

Settings.js
Just put some configuration variables here, can be easily extended by adding a new variable as you need it.

FontManager.js
Font Manager is class devoted to register and draw texts on the canvas this is a list of methods that you can find inside:

```
fontList;
```
Used to hold all the fonts registered for the engine. By default starmap is already loaded.
```
gameVariables;
settings;
```
These two are references for the gameVariables from engine.js and settings Class from settings.js

```
constructor(gameVariables, settings) {}

addFont(name, style) {}
```
This is used to register a font in the engine with this format:
```
this.fontManager.addFont('starmap', { color: 'white', size: 18 });

drawText(canvas, maxWidth, text, x, y, fontName, color, stroke) {}
```
Draw text on the canvas with some caveats:

1) Given **maxWidth** you can define how much the text must spread horizontally before breaking to next line
2) **text** can contain metacharacter to allow for styled text using §r, §b, §i or §0-9 to go from regular to bold to italic and to change from **color** to some predefined one defined in settings.js
3) **stroke** can be omitted as by default its value is false

An example of using **drawText** is the following:
```
this.fontManager.drawText(this.gameCanvas, 200, name, 310, 180, 'starmap',
'white');
```

Following are some private methods included in drawText.

```
drawOrStroke(ctx, text, x, y, fill) {}

drawStyledText(ctx, text, x, y, fontName, size, fill) {}

transformVariables(text) {}

getTriggerValue(match) {}
```

...

Interactables

Interactables are the core of the game engine as they define how elements interact to the player and to each other in order to unlock new pieces of the scenes and help the player complete the story. All of the interactables can be created inside the JSON maps. Here is a list of all the interactable already created, at the end will see how it is possible to define a new interactable.

Look interactable

This is the interactable that can be used to describe something in the scene. Is one of the most simple. This is the structure:

```
{
 "type": "look",
 "linked": "<object_name/character_name>",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "messages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
  ]
}
```

Messages can be 1 or n. This **interactable** as **no effect** apart defining a dialog.

Dialog interactable

This interactable is similar to the look interactable apart from the fact that it is explicit to the player that it is a dialog between the main character and an NPC. Structure:

```
{
 "type": "dialog",
 "linked": "<object_name/character_name>",
```

```
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "messages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
  ]
}
```

Object interactable

This is used to define an object that can be taken by the user by clicking over it. Structure:

```
{
 "type": "object",
 "description": "String",
 "linked": "<object_name/character_name>",
 "sound":  "(optional) filename.extension",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "messages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
 ]
}
```

Interact interactable

Interact is used to describe elements which change state or trigger changes in the scene or are useful for later combining actions. Structure:

```
{
 "type": "interact",
 "linked": "<object_name/character_name>",
```

```
 "trigger": "toggle/0/1",
 "sound":  "(optional) filename.extension",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "messages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
 ]
}
```

Combine interactable

Combine allows to verify if the user possesses some objects or activated some triggers to get a specific result. Structure:

```
{
 "type": "combine",
 "linked": "<object_name/character_name>",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "conditions": [
   { "object": { "name" : "<object_name>", "value": 1 } },
   { "trigger": { "name": "eri", "value": 1 } },
   ...
 ],
 "result": [
   { "trigger": 1 },
   { "object": { "name": "<object_name>", "description": "String" } },
   { "sound":  "filename.extension" }
   { "wingame": 1 }
   ...
 ],
 "notMetMessages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
```

```
],
 "completedMessages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
 ]
}
```

Question interactable

Used to define a question with several answers (max 4) and a reward along with different messages for the different results. Structure:

```
{
 "type": "question",
 "linked": "<object_name/character_name>",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "question": {
   "name": "String",
   "text": "String",
   "answers": [
     { "text": "String", "valid":  0/1 },
     ...
   ],
   "result": [
     { "trigger": 1 },
     { "object": { "name": "<object_name>", "description": "String" } },
     { "sound":  "filename.extension" }
     { "wingame": 1 }
     ...
   ]
 },
 "messages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
 ],
 "goodAnswerMessages": [
```

```
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ],
  "wrongAnswerMessages": [
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ],
  "completedMessages": [
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ]
}
```

Minigame interactable

This is used to launch a minigame. Structure:

```
{
  "type": "minigame",
  "linked": "<object_name/character_name>",
  "game": "<minigame_id>",
  "x": number,
  "y": number,
  "width": number,
  "height": number,
  "conditions": [
    { "object": { "name" : "<object_name>", "value": 1 } },
    { "trigger": { "name": "eri", "value": 1 } },
    ...
  ],
  "result": [
    { "trigger": 1 },
    { "object": { "name": "<object_name>", "description": "String" } },
    { "sound":  "filename.extension" }
```

```
    { "wingame": 1 }
    ...
  ],
  "notMetMessages": [
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ],
  "completedMessages": [
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ],
  "retryMessages": [
    {
      "type": "self/other",
      "name": "String",
      "portrait": "<main/object_name/character_name>",
      "text": "String"
    }
    ...
  ]
}
```

Teleport interactable

This interactable is used to go from one scene to another directly. Is perfect for cutscenes.
Structure:

```
{
  "type": "teleport",
  "linked": "<object_name/character_name>",
  "x": number,
  "y": number,
  "width": number,
  "height": number,
  "goTo": <mapNumber>,
  "spawn": [<numberX>, <numberY>]
}
```

Exit interactable

Same as the teleport interactable, the exit interactable is used for transition between game maps and some conditions can be applied to prevent teleporting before they are met. Structure:

```
{
 "type": "exit",
 "linked": "door",
 "linked": "<object_name/character_name>",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "goTo": <mapNumber>,
 "spawn": [<numberX>, <numberY>],
 "conditions": [
   { "object": { "name" : "<object_name>", "value": 1 } },
   { "trigger": { "name": "<trigger_name>", "value": <0/1> } }
 ],
 "notMetMessages": [
   {
     "type": "self/other",
     "name": "String",
     "portrait": "<main/object_name/character_name>",
     "text": "String"
   }
   ...
 ]
}
```

Open Url interactable

Open url allows to open a different url, Beware this is a destructive action so can be called just once. The idea is to use it for going to a crm form. Structure:

```
{
 "type": "openurl",
 "linked": "<object_name/character_name>",
 "x": number,
 "y": number,
 "width": number,
 "height": number,
 "url": "https://www.google.it"
}
```

Map JSON file

Following is the structure of a game Map file. The file is written in JSON format. Here you can basically define your entire game. Structure:

```
{
 "interactables": [...],
```

```
 "characters": [...],
 "objects": [...],
 "staticTexts": [...],
 "bg_R_offset": 0,
 "bg_M_offset": 0,
 "bg_F_offset": 0,
 "startingPoint": [number, number],
 "startingMusic": "filename.mp3",
 "walkableOffset": 0,
 "name": "String"
}
```

Main properties

These properties must always be there:

```
"bg_R_offset": number (y offset of bg),
"bg_M_offset": number (y offset of bg),
"bg_F_offset": number (y offset of bg),
"startingPoint": [number, number],
"startingMusic": "filename.mp3",
"walkableOffset": number (y offset of player),
"noplayer": true (optional: for cutscenes),
"name": "String"
```

Static texts

(OPTIONALS) Some texts displayed over the screen

```
{
  "name": "String (unique id)",
  "width": number,
  "text": "String",
  "color": "html5 valid color, eg: white, black, #445566, rgba(1,1,1,0.3)",
  "font": "String (any registered font name)",
  "x": number,
  "y": number
}
```

Objects

Static elements on the screen. Structure:

```
{
     "name": "String (unique id)",
     "x": number,
     "y": number
}
```

Characters
Defines all the animated sprites here, this area can also be cleverly used for screen animations, using “big sprites”. Structure:

```
{
     "name": "String (unique id)",
     "sWidth": number (the single frame width),
     "frames": number (frames),
     "x": number,
     "y": number
}
```

Interactables
All the bounding boxes with special interaction available on the map, please refer to the previous chapter for detailed information.

Minigames

Minigames are real javascript games that lives on their own, but, in order to be called inside the main game loop they must adhere to some interfaces and conventions.

The minigames plugin interface and registration
First thing first: in order to create a minigame you must create a javascript file that adhere to this interface. In case you need external libraries invoke them in the index.html file before adding the main js file (follow the example at the beginning of this manual).

The structure for the main file is the following:

```
constructor(
    gameCanvas,
    gameWidth,
    gameHeight,
    settings,
    soundSystem,
    spriteManager,
    fontManager,
    mapManager,
    mouseManager,
    interactableManager,
    gameStatus,
    gameVariables) {}

start() {}

animate(time) {}

draw(time) {}
```

```
processMovement(canvasX, canvasY) {}

processClicks(canvasX, canvasY) {}
```

Everything is passed to the minigame in order to access all the functions and statuses of the main game. It is NOT fundamental to use all of these methods as they are mainly here to manage games done in canvas mode, BUT `start()` is mandatory in order to make the game start, we'll see how.

```
animate(time) {}
```
Animate here is used when you need to define an animation cycle just like in the main game, you also have access to timer and the game tick is already taken care of; this method is called already at the right speed.

```
draw(time) {}
```
Used if you need to draw anything on the main canvas. Dpi and clear buffer are already taken care of. You can also have access to every spriteManager and fontManager's method you need to help you out.

```
processMovement(canvasX, canvasY) {}
processClicks(canvasX, canvasY) {}
```
Here you can manage click and mouse's movements on the canvas. Please note that as said before, this is not necessary as you can develop a minigame by other means and use it inside the main game, the FalloutMinigame is an example of this.

How to integrate a minigame
To integrate a minigame you need to **put all of your code, images, sound and music inside a folder** and put that folder in **assets/minigames**. If you need CSS please add it **after engine css in style.css**.

If you manage to do so, please consider **packing your game in a single js file**.
Then here you can see how to add custom CSS rules; just put them in **src/css/style.css**:

```css
@font-face { font-family: starmap;  src: url('../assets/fonts/starmap.ttf'); }

html { background: black; }
body { height: 100%; }
canvas { width: 100%; height: 100%; touch-action:none; user-select:none; }
div.main-game { background: black; padding: 0; margin: auto; display: block; width: 1000px; height: 480px; }

/* ======================================================== */
/* Minigame addons: you can add minigames specific css here */
/* ======================================================== */

/* ========== FALLOUT TERMINAL HACKING MINIGAME ========== */
#terminal {
    width: 750px; height: 460px; background-image: url('../assets/minigames/fallout-minigame/img/bg.png');
    font-family: starmap; font-weight: normal; color: #33dd88; font-size: 12pt;
    overflow: hidden; z-index: 10; position: absolute; left: 50%; margin-left: -375px; top: 87px;
}
#terminal-background {
    background-image: url('../assets/minigames/fallout-minigame/img/monitorborder.png');
    width: 930px; height: 700px; z-index: -20; margin: 0 auto; position: relative;
}
#terminal-interior { position: relative; width: 100%; height: 100%; z-index: 11; }
.column { margin-top: 30px; height: 350px; top: 90px; margin-left: 20px; float: left; }
.pointers { width: 51px; }
.words { width: 171px; }
.character { padding-left: 2px; padding-right: 2px; cursor: pointer; }
.character-hover { background-color: #33dd88; color: #112211; }
```

Then you need to **add dependencies to the index.html**:

```html
<script src="js/mapManager.js"></script>
<script src="js/minigame.js"></script>
<!-- Minigames here -->
<script src="assets/minigames/fallout-minigame/fallout-minigame.js"></script>
<!-- ============== -->
<script src="js/engine.js"></script>
```

And register the minigame before launching the engine. Just use a unique name to identify it:

```html
<script>
    const game = new beAdventurousEngine("game");
    game.registerMiniGame("FalloutMinigame", FalloutMinigame);
    game.start(0);
</script>
```

Two thing are mandatory and must taken care of in your code: Starting and Finishing conditions.

Starting a minigame

For an example this is the content of the fallout minigame which uses standard html to define the game screen. So in order to make it work with the current system a couple of commands must be issued, which in part are mandatory in every minigame. These are issued in `start()`.

```
// Hide current canvas and stop current music
this.currentMusic = this.gameVariables.currentMusic;
this.soundSystem.stopBackgroundMusic();
this.gameCanvas.parentElement.style.display = "none";
this.gameStatus.minigameWin = undefined;
```

As can be seen we remove and save current music from the main game, make the original game invisible (which also stops the rendering process from using CPU and GPU) and most importantly we **set** `this.gameStatus.minigameWin` to `undefined`. This is mandatory as

guarantee that the engine **recognizes being in minigame mode**. After tis you can proceed coding your game as you see fit.

Ending a minigame

Minigames must have 2 outcomes: winning or losing, so in order to make everything work as expected in the example minigame two methods are created which are called when a player win or loses the minigame inside the game's loop.

```
Success() {
    this.UpdateOutput("Access granted.");
    // Do success here
    this.gameStatus.minigameWin = true;
    this.gameVariables.currentMusic = this.currentMusic;
    this.soundSystem.playBackgroundMusic(this.currentMusic);
    this.gameCanvas.parentElement.style.display = "block";
    this.destroyGame();
}
```

When the player wins, we set some important variables:

- `this.gameStatus.minigameWin = true` to ensure we return to the main game in winning mode. This will lead to winning messages and results defined in the map's interactable.
- `this.gameCanvas.parentElement.style.display = "block"` we show the main canvas again.
- We remove all the created html for the minigame from index.html. Please note that only the **gameStatus** variable is mandatory, the rest is done to accommodate the nature of this particular game which draws in a different way from the main game.

```
Failure() {
    this.UpdateOutput("Access denied.");
    this.UpdateOutput("Lockout in");
    this.UpdateOutput("progress.");
    // Do failure here
    this.gameStatus.minigameWin = false;
    this.gameVariables.currentMusic = this.currentMusic;
    this.soundSystem.playBackgroundMusic(this.currentMusic);
    this.gameCanvas.parentElement.style.display = "block";
    this.destroyGame();
}
```

The failure condition is similar but we set `this.gameStatus.minigameWin = false`, this way we enter in losing mode the main loop and the failure messages will be parsed from the map.

Fallout Mini Game example

Included in the source code there is also the fallout hacking minigame, modified to be used inside the engine. The code is very small (500 lines) and can be used as an interesting example

of how a minigame can be done diverging from the original engine system. What is important is to always show one thing: the minigame or the main game.

Tutorial

Simple step by step guide to create a map.

Start by downloading the source code from the repo:
https://github.com/urz9999/beAdventureEngine/

The source code can be placed anywhere on your computer. Then go to the src folder.

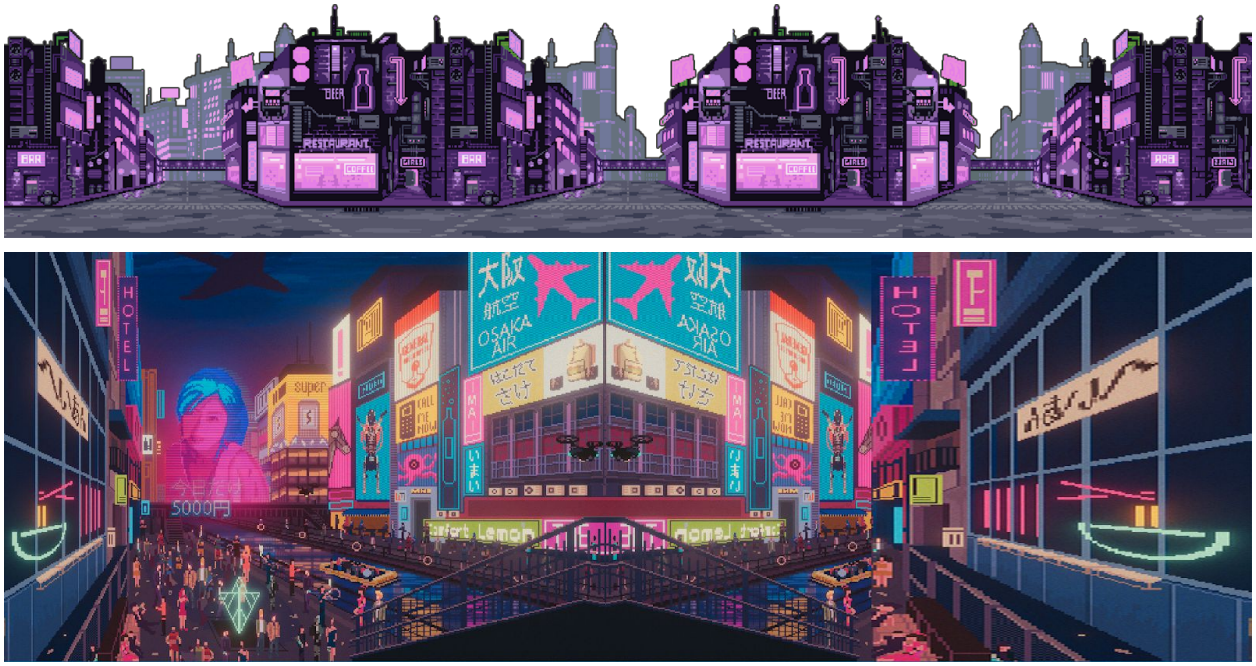
Prepare the project

You can keep or remove all images in **assets/images/ui/maps** and in **assets/images/ui/objects** and **assets/images/ui/pgs**. Keep the images in the **menu** as they can be replaced but give exact dimension templates. You can also keep the original graphics for the tutorial.

Clear the minigame directory if you don't plan to add the default minigame to your demo.

Go to **index.html** and if you have removed the minigame, remove the .js relative to the minigame. Now you have a cleaned project to start creating your own map to test.

Create a new map
To create a map take the default code in the **Main properties**, and be sure to create 3 backgrounds and name them accordingly. Place them inside **assets/images/maps/map1/** calling them as **map1_R.png**, **map1_M**.png, **map1_F.png**, where R is Rear, M is Medium and F is Front. You can use these 3 for the example if you want:



Create **map1.json** inside **assets/map** and copy the following structure:

```
{
 "interactables": [],
 "characters": [],
 "objects": [],
 "staticTexts": [],
 "bg_R_offset": 0,
 "bg_M_offset": 0,
 "bg_F_offset": 0,
 "startingPoint": [200, 200],
 "startingMusic": "demo.mp3",
 "walkableOffset": 0,
 "name": "My fist map"
}
```

Prepare a music you like, copy it and name the file **demo.mp3**, put it in **assets/music**.

We also need the main character as this is a map with the player in. Please go to the **assets/images/ui/pgs** and add **main.png** and **mainFace.png**; you can always use these default images:

Go to **index.html** and modify the command *game.start* to `game.start(1);` in the script portion of the file (line 37).

As you are debugging your first game it's better for now to toggle **fullscreen** off at the beginning of the game to help you out with **x,y** coordinate management. Go to **settings.js** and turn the **fullscreen variable** to **false** if you want the **debug** to **true**.

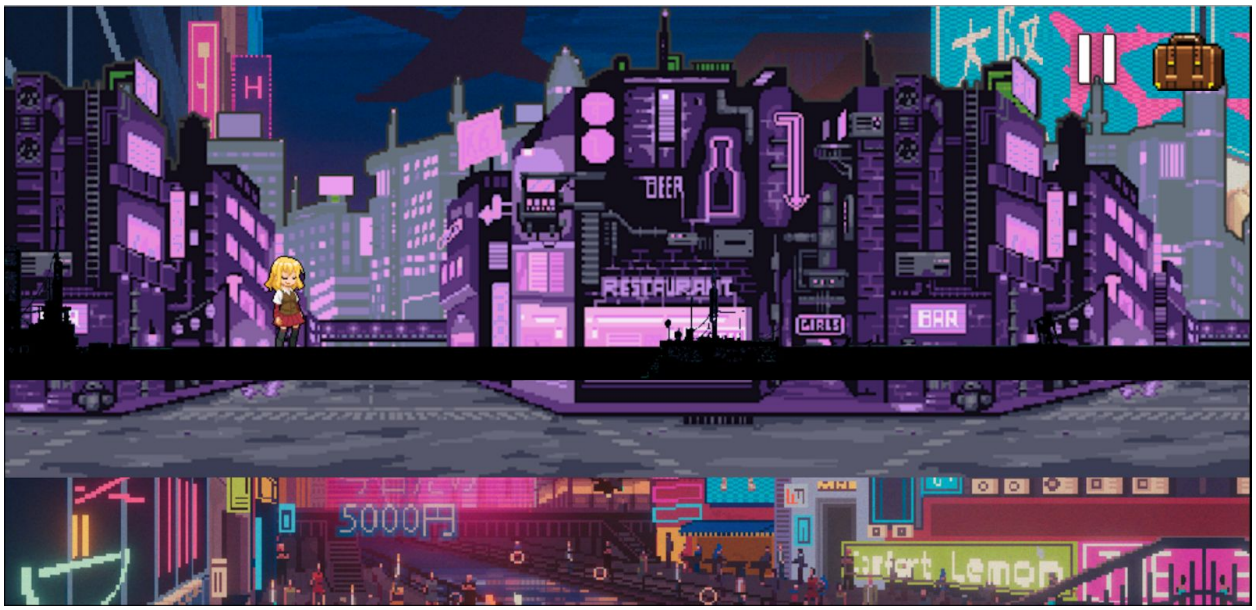
```
constructor() {
    this.fps = 30;
    this.gameTick = 6000;
    this.creditMap = 99; // Set here what map to use for credit

    this.debug = true;
    this.fullscreen = false;
```

Go to your **terminal** and navigate to the **src folder** of the project then type this command to open a **Python web server on port 8080** (be sure to have Python installed):

```
python3 -m http.server --cgi 8080
```

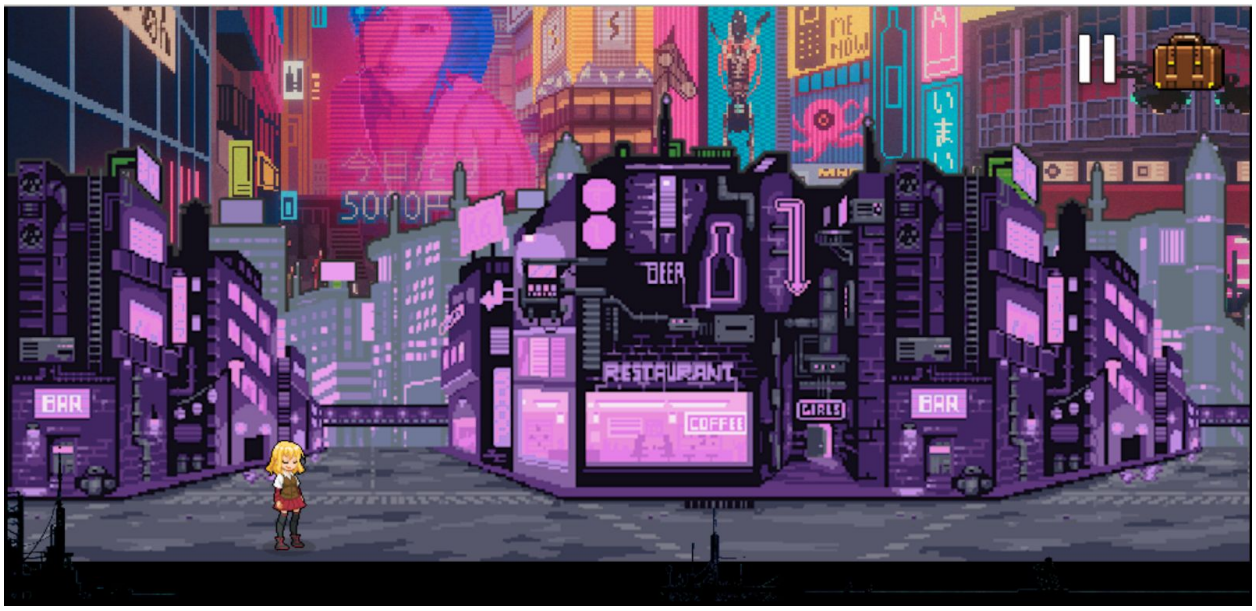
Now go to your browser of choice (preferred ones are still Chrome and Firefox though) and hit **localhost:8080**, press enter and see if the map is showing. As you'll see the map is quite messy but we can set it correctly.



To correct the position of all elements simply change the Y offset of the backgrounds and the player like this:

```
"bg_R_offset": -250,
"bg_M_offset": 70,
"bg_F_offset": 200,
"startingPoint": [200, 350],
```

Basically we are requesting to shift the elements until we see them fitting the screen. Normally you'll do this by taking references from a drawing software like Paint, Gimp or Photoshop opening a new file and making the dimension of the canvas like the dimension of the game when not in **fullscreen** (the default value is **1000x450**, but you can change these values from the **style.css** file). Another solution is simply hitting F5/Cmd+R on the browser and going the trial-and-error way. This is the result:



Add sprites to interact with

Much better! Now let's add a static and a dynamic sprite on the map to interact with. Download and put this sprite in the **assets/images/ui/objects** folder, naming it **money.png**:

And these 2 in the **pgs** folder, naming them **eri.png** and **eriFace.png**:

Remember that every time you create an animated sprite you can always generate a face for the dialogs by creating a **128x128** portrait with the same name and the Face suffix.

Let's add these two elements in the map. Add this code to the **object** array in **map1.json**:

```
{
 "name": "money",
 "x": 400,
 "y": 405
}
```

*Note that **money** is the same name as **money.png**, this is fundamental to make the engine able to retrieve the correct sprite, remember?*

Now for the **character** array add this one:

```
{
 "name": "eri",
 "sWidth": 76,
 "frames": 6,
 "x": 890,
 "y": 353
}
```

A couple of things to note here:

`"sWidth": 76` here is used to describe the width of a **single frame** of **eri.png**, you can get this information from any paint tool.

`"frames": 6` Is the total number of frames in the image.

The rest is similar to the object code. Let's see the result.

The objects are on the scene, now we can make them interactive by adding 2 interactables, for example we can pick up the money with an **object interactable** and we can talk with the character with a **dialog interactable**. Let's add them.

Add interactable to the scene

To add a simple object interactable copy this code in the **interactable** array in the **map1.json** file.

```
{
 "type": "object",
 "description": "a little pile of money laying on the ground",
 "linked": "money",
 "x": 400,
 "y": 405,
 "width": 27,
 "height": 24,
 "messages": [
   {
     "type": "self",
     "name": "Fiona",
     "portrait": "main",
     "text": "Money! Let's take it!"
   }
 ]
}
```

In this case we can refer to the **interactable section of this manual** for more infor but a couple of notes:

`"linked": "money"` this refers to the name of the image/object, is always the same.

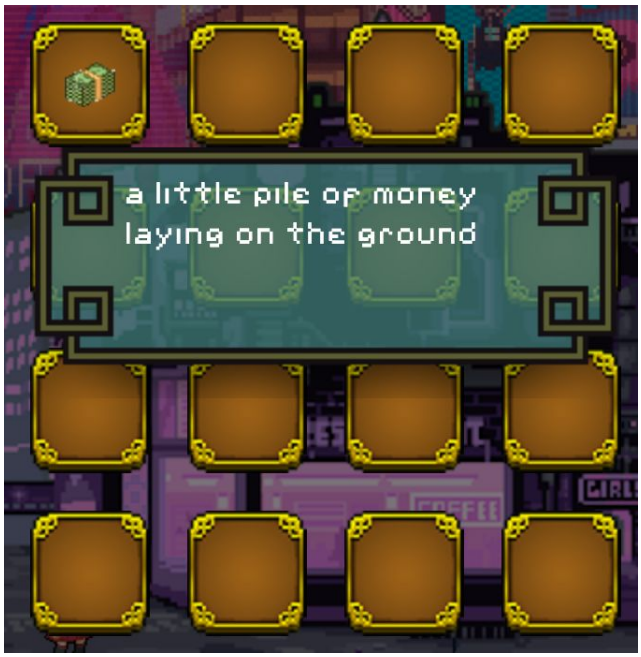
```
"x": 400,
"y": 405,
"width": 27,
"height": 24,
```

These values are retrieved both from the object's code (x and y) and by going to the image property for example using mouse dx button and check in **properties (Windows)** or **Get Info (MACOS)**. Look for the width and height of the image.

```
"messages": [
   {
     "type": "self",
     "name": "Fiona",
     "portrait": "main",
     "text": "Money! Let's take it!"
   }
 ]
```

If this array is not empty a simple pickup message can be created. The structure is described both in **dialog interactable** and in **object interactable** itself, always in this manual.

Let's see if the object can be picked up now. Also if you have selected **debug = true** in the setting.js file, you'll see a bounding box around the money sprite.



As you can see the bounding box is there, you can click on the image and a dialog will appear, the object disappears and now you have it in the inventory. Great! Creating and picking up objects is very important to unlock complex scenes with the **combine interactable**.

Now for the character:

```
{
 "type": "dialog",
 "linked": "eri",
 "x": 890,
 "y": 353,
 "width": 76,
 "height": 80,
 "messages": [
   {
     "type": "self",
     "name": "Fiona",
     "portrait": "main",
     "text": "Good morning!"
   },
   {
     "type": "other",
     "name": "Eri Kasamoto",
     "portrait": "eri",
     "text": "Hello! how are you?"
   },
   {
     "type": "self",
     "name": "Fiona",
     "portrait": "main",
     "text": "Fine thanks! See you around!"
   }
 ]
}
```

The code is straightforward and must be pasted after the object interactable following JSON standard, the properties are similar and the messages part also, just a bit longer. Note that we use **self/other** to describe who is talking and **main/<name>** to define which portrait we need to use.

Here is the dialog in action. Note that if you click on the character, your main sprite will move towards the interactable and the engine will take care of bigger than canvas scenes, movement, collision, and activating the dialog.

This of course is a simple procedure to add things to the map. If you image to iterate these processes on more than one map and by making clever use of all interactable you can create simple or complex games with ease. Please also refer to the demo included to have a glance of all types of interactables available to you.

Demo

To play the demo just download the provided source and play it on a local server. If you have python an easy way is to go to the **src folder and type this command** in the terminal (on MACOS and Linux/Ubuntu this can be done by default):

```
python3 -m http.server --cgi 8080
```

Helpful resources

https://ezgif.com/gif-to-sprite to convert gif into sprite sheets compatible with the engine
https://www.cursor.cc to convert images to cursors compatible with the game
https://www.online-convert.com/ to convert ogg to wav or mp3 format
https://opengameart.org to get many valuable assets for free
https://www.spriters-resource.com/ to get tons of RIPS, but beware that on these copyright may still exists