dim-boilerplate Documentation

Release 0.1

Peter Parente

Sep 27, 2017

Contents

The dim-boilerplate is a starting point for building web based adventure games using the same basic speech, sound, image, text, and keyboard support as Descent Into Madness. This documentation describes how you can build-out the template to meet your game needs in successively more flexible and interactive but complex ways.

Contents:

CHAPTER 1

Get Started

Understand the Capabilities

The dim-boilerplate project provides the following features out-of-the-box:

1. Declarative JSON markup for defining game world scenes, items, and events.
2. Exploration of the game world via arrow key or gesture-driven menus.
3. Game world interactions including examining, moving, taking, and using.
4. Game world puzzles involving sequencing and timed reactions.
5. Visual backgrounds and text descriptions.
6. Aural text-to-speech synthesis, voice actor narration, sound effects, music, and ambiance.

Keep in mind that the project is a template for writing adventure games, not a strict framework or engine. You should drop, replace, extend, or otherwise hack any of the included components to suit your needs.

Clone the Boilerplate

The boilerplate project is available on GitHub. Clone it with the following command:

```
git clone git://github.com/parente/dim-boilerplate.git
```

Understand the Contents

The boilerplate project has the following general layout (not all files are shown):

```
.
- docs/                         # source for this documentation
- dev/                          # development and deployment scripts
- webapp/
    - css/                      # game visual stylesheets
    - data/                     # game world assets
    |   - world.json            # game world definition
    - favicon.ico               # game browser bar icon
    - img/                      # game iOS startup and touch icons
    - index.html
    - js/                       # game engine
    |   - dim/
    |   |   - aural/
    |   |   - controllers/      # game interaction controllers
    |   |   - events.js
    |   |   - input.js
    |   |   - main.js
    |   |   - pump.js
    |   |   - topic.js
    |   |   - visual.js
    |   |   - world.js
    |   - vendor/               # dependencies
    - robots.txt
```

Setup for Development

Push the boilerplate into a new git repository where you will build your game. Alternatively, copy at least the dev and webapp folders to a new directory in the version control system of your choice.

The boilerplate has no web server-side requirements. You can simply put the webapp folder in a web accessible directory (e.g., ~/Sites on OS X, ~/public_html on Linux) and visit the index.html page in a browser. As you make build-out your game, you can simply refresh your browser to immediately see the impact of your changes.

Currently, the boilerplate supports the following browsers:

1. Chrome on any platform
2. OS X Safari 6.x

Note: Ultimately, the boilerplate should be compatible with any browser that supports the HTML5 Web Audio API.

Build Your Game

Draft the storyline of your adventure game. Then try to answer these questions.

1. What is the goal of the game?
2. What locations will the player visit?
3. What other people and things exist in the world?
4. What interactions will the player have?
5. What challenges will the player face?

Now modify the boilerplate to implement your game ideas. Work through the following levels of modification in the order given, only continuing to higher levels if you cannot accomplish what you need in a lower level.

- *Level 1* - Define your game scenes, items, and events in the game world JSON file using text, sounds, speech, and images.
- *Level 2* - Create new JavaScript controllers to enable new user actions within the game world.
- *Level 3* - Modify the input capture, aural rendering, and visual rendering JavaScript code to support new means of interaction with the user.

CHAPTER 2

Level 1: Define the World

Much of the effort in creating a game is spent defining the scenes, items, and events in the world JSON file. The world.json file resides in the webapp/data directory. The following sections describe its format by way of an example game in which the player needs to cook and serve dinner.

The World Array

The top element of the JSON file is the world array. The array contains objects describing elements of the game world. The object properties declared in the JSON file represent the state of the elements at the start of the game. As a player interacts with the game world, the contents of the original world array and its objects change.

```
[
    {
        /* a game world element */
        "type": "scene",
        "id": "garage"
    },
    {
        /* a second game world element */
        "type": "item",
        "id": "car"
    }
]
```

Every object in the world must have a *type* property and many must have an *id* property unique within its type. The following sections detail the different types and their associated properties.

Note: The JSON snippets below contain /* comments */ for documentation purposes. They are not allowed in your actual JSON file according to the JSON standard. Including them may cause your game to fail to load.

Player

The player object represents the state of the player. The player may hold zero or more items and reside in one scene at a time. The world array may contain one and only one player object.

```
{
    "type": "player",
    /* items the player is carrying */
    "items": ["groceries"],
    /* scene containing the player */
    "scene": "garage"
}
```

Scenes

A scene object describes a location in the game world. A scene may contain zero or more items and adjoin zero or more other scenes. A scene may have visual and aural properties describing how to appears to the player. The world array may contain any number of scenes.

```
{
    "type": "scene",
    /* unique scene id */
    "id": "kitchen",
    /* ids of items in the scene */
    "items": ["stove", "dishwasher"],
    /* ids of scenes connected to this one */
    "adjoins": ["living room"],
    /* visual properties of the scene */
    "visual": {
        /* text name */
        "name": "Kitchen",
        /* text description */
        "description": "A modern kitchen with a stainless steel stove, dishwasher,␣
↪and refrigerator",
        /* background image */
        "backdrop": "image://backdrop/kitchen.png"
    },
    /* aural properties of the scene */
    "aural": {
        /* spoken name */
        "name": "sound://speech/kitchen",
        /* spoken description */
        "description": "sound://sound/kitchenDescription"
        /* looping ambiance */
        "backdrop": "sound://music/cookingMusic"
    },
    /* (optional) controller that activates when the user enters the scene */
    "controller": "dim/controllers/meta/explore"
}
```

If the scene does not define a controller, the one specified in the *default object* is used instead. The aural properties can contain *media URIs* pointing to sound files or plain text to synthesize as speech.

Items

An item object describes an inanimate object in the game world. An item may have zero or more properties. An item may have visual and aural properties describing how to appears to the player. The world array may contain any number of items.

```
{
    "type": "item",
    "id": "stove",
    "visual": {
        "name": "Stove",
        "description": "It's a gas stove. It's off."
    },
    "aural": {
        "name": "sound://speech/stove",
        "description": "sound://speech/stoveOffDesc"
    },
    "properties": ["useable"]
}
```

Controllers give meaning to item properties. They have no impact on the game world on their own.

Events

An event object describes changes made to the game world upon some player interaction with the world. It also describes how to report those changes to the player. The world array may contain zero or more events.

An event has three main properties:

1. *on*, an array of strings which state when the event fires
2. *exec*, an array of actions which change the state of the game world when the event fires
3. *report*, an array of visual and aural information to report to the player about the event

For example, the following example event fires when the player uses the stove item. When it fires, it disables the event from firing again and enables the *panToStove* event. It also changes the description of the *stove* object to explain that the stove is now lit. Finally, the event object visually describes what happens when the player uses the stove and narrates it as well.

```
{
    "type": "event",
    "id": "lightStove",
    /* conditions for triggering this event */
    "on": [
        "use",
        "stove"
    ],
    /* actions to take on the game world state when the event fires */
    "exec": [
        {
            /* sets a property arg0 to value arg1 */
            "action": "set",
            /* here, set the event.lightStove.disabled to true */
            "args": [
                "event.lightStove.disabled",
                true
            ]
```

```
        },
        {
            "action": "set",
            "args": [
                "event.panToStove.disabled",
                false
            ]
        },
        {
            "action": "set",
            "args": [
                "item.stove.visual.description",
                "The gas stove is hot and ready to cook."
            ]
        },
        {
            "action": "set",
            "args": [
                "item.stove.aural.description",
                "sound://speech/stoveOnDesc"
            ]
        }
    ],
    /* report to give after making the changes to the world */
    "report": [
        {
            "description": "You light the stove.",
            "narration": "sound://speech/lightStove"
        }
    ]
}
```

The ability of events to affect the game world is limitless. The flexibility of the *on*, *exec*, and *report* properties make this possible. The next sections describe these properties in more detail.

On - Event Triggers

The *on* array supports any number of strings of any value. All but two reserved strings are meaningless on their own. The rest are given meaning by the *game controllers*.

Upon certain user interactions (e.g., taking an item, examining a scene, shooting a bad guy), a controller may ask the game world to evaluate its events to see if one matches an array of strings provided by the controller. The world returns any event object that has an *on* array matching the controller's array of strings. The controller may then fire any of the returned events.

For example, if a controller does the following:

```
var events = world.evaluate('use', 'stove');
events.fires();
```

the event defined above will fire.

Two strings have special meaning in the *on* array. A single star, *, matches any single string element in a controller provided array. For instance, if the world.json includes this event:

```
{
    "type": "event",
```

```
    "on": ["use", "*"]
}
```

then it will match the first two controller calls below, but not the third:

```
world.evaluate('use', 'stove');
world.evaluate('use', 'sink');
world.evaluate('use', 'dishwasher', 'quickly');
```

Two stars, **, at the end of the *on* array match any number of string elements. For instance, if the world.json includes this event:

```
{
    "type": "event",
    "on": ["use", "**"]
}
```

then it will match all three controller calls above.

Note: The boilerplate has controllers that evaluate and fire events matching the following patterns. If you want to support other events, add them to the world.json and write controllers to trigger them. See the *controllers* section for details.

- ["use", "<item id>"]
- ["use", "<item id>", "<other item id>"]
- ["move", "<scene id>"]
- ["examine", "<item or scene id>"]
- ["take", "<item id w/ takeable property>"]

Exec - Event Consequences

The *exec* array supports a variety of actions that modify the game world when the event fires. Each action takes a set of arguments which may have simple values, refer to names of properties in the world, or values of argument properties. The next sections explain the actions supported by the boilerplate, their arguments, and sample uses by way of example.

Note: The following sections define the actions as functions. In practice, they are used declaratively in the world JSON.

---

activate(*controller_module*, *arg1*, *arg2*, ...)

The activate action transfers input handling to another controller from the current one.

Arguments

- **controller_module** (*string*) – AMD path of the controller module (see example)
- **args** (*any*) – Optional arguments to pass to the controller's initialize function

For example, the following action activate the *cook* controller providing it with the arguments *eggs* and *cheese*.

```
{
    "action": "activate",
    "args": ["dim/controllers/puzzle/cook", "eggs", "cheese"]
}
```

append (*target_ref*, *value*)

The append action adds a value to the end of an array declared in the world JSON.

Arguments

- **target_ref** (*string*) – Name of the target array, referenced by its *type.id.property*
- **value** (*any*) – Value to append to the target array

For example, the following action adds the item with the *pan* ID to the array of player items (i.e., adds the pan to the player inventory).

```
{
    "action": "append",
    "args": ["player.items", "pan"]
}
```

del (*target_ref*)

The del action deletes a property declared in the world JSON.

Arguments

- **target_ref** (*string*) – Name of the target property, referenced by *type.id.property*

For example, the following action deletes the *exec* property from the *panToStove* event (i.e., prevents the event from changing the game world again).

```
{
    "action": "del",
    "args": ["event.panToStove.exec"]
}
```

remove (*target_ref*, *value*)

The remove action deletes a value from an array of values declared in the world JSON.

Arguments

- **target_ref** (*string*) – Name of the target array, referenced by its *type.id.property*
- **value** (*any*) – Value to remove from the target array

For example, the following action removes the item with the ID *pan* from the *cupboard* scene.

```
{
    "action": "remove",
    "args": ["scene.cupboard.items", "pan"]
}
```

set (*target_ref*, *value*)

The set action stores a value in a property declared in the world JSON.

Arguments

- **target_ref** (*string*) – Name of the target property, referenced by its *type.id.property*
- **value** (*any*) – Value to set for the property

For example, the following action changes the visual description of the *pan* item to state that the pan is hot.

```
{
    "action": "set",
    "args": ["item.pan.visual.description", "The pan is blazing hot!"]
}
```

Report - Event Explainations

The *report* array contains the information to report to the player when the event fires. The array consists of zero or more objects to be reported in sequence. The objects have keys representing output *channels* paired with the values to report on those channels. The channels are declared and configured world JSON (see *Default*). The boilerplate defines the following by default:

title Text shown at the top of the default game UI

description Text shown in the center of the default game UI

backdrop Image rendered in the background of the default game UI

narration Speech reported on an audio channel of the default game UI

sound Sound played on an audio channel of the default game UI

ambience Repeating sound or music played on an audio channel of the default game UI

For example, the following report shows and narrates the event of putting eggs in the pan on the stove. While the narration describes the eggs cooking, a short sound of sizzling eggs plays. After the visual description appears, and the narration and sound conclude, the report plays a second short sound of eggs sizzling more loudly.

```
[
    {
        "description": "The eggs start to cook.",
        "narration": "sound://speech/eggsCook",
        "sound": "sound://sound/eggsCook"
    },
    {
        "sound": "sound://sound/eggsCookMore"
    }
]
```

Any fields ommitted from the report are left unchanged. For example, if the visual *title* was previously set to "Kitchen" by an earlier report, it remains untouched as "Kitchen" when this event fires.

The user input event may interrupt the current report in progress. When this happens, all other pending reports (e.g., the second object in the example array above) are also skipped to allow a report about the latest user action to start immediately. Typically, interruptions only impact aural channels as visual channels complete their reports almost immediately (i.e., they appear on the screen).

Templates

All of the *exec* and *report* examples given so far use fixed references and values. None of them vary based on the game state. Though simple and common, these action arguments are limiting and do not scale well. For instance, the use of a *pot*, a *pan*, or a *wok* on the *stove* requires three separate event objects when using hardcoded strings alone.

The special *on* array arguments * and ** provide a foundation for solving this problem. Recall that these arguments match arbitrary strings provided by controllers. If a controller provides a game world object in place of a string, the object's string ID is used to match events. If a match occurs, the event is able to reference any game world object that

participated in the match. *Templates* in the event *exec* and *report* fields can include values read from matched objects, customizing the event output and actions based on the arguments provided.

For instance, consider this event which allows the player to use any item with the *stove*. Various *exec* and *report* fields in this event have templates refering to the object matched by the * in the *on* array. These templates allow this single event object to handle the use of a *pot*, *pan*, *wok*, or any other item a controller cares to interface with the *stove*.

```
{
    "type": "event",
    "id": "itemToStove",
    /* use X with the stove, where X is an item provided by a controller */
    "on": [
        "use",
        "*",
        "stove"
    ],
    "exec": [
        {
            "action": "remove",
            "args": [
                "player.items",
                /* template referring to the on[1] matched object's id */
                "{{{args.1.id}}}"
            ]
        },
        {
            "action": "append",
            "args": [
                "scene.kitchen.items",
                /* template referring to the on[1] matched object's id */
                "{{{args.1.id}}}"
            ]
        },
        {
            "action": "append",
            "args": [
                /* template referring to the on[1] matched object's id */
                "item.{{{args.1.id}}}.properties",
                "hot"
            ]
        }
    ],
    "report": [
        {
            /* template referring to the on[1] matched object's visual name */
            "description": "You place the {{args.1.visual.name}} on the hot stove.",
            /* template referring to the on[1] matched object's id */
            "narration": "sound://speech/{{{args.1.id}}}ToStove"
        }
    ]
}
```

In general, event templates follow the format below:

```
{{{args.indexOfOnArrayElement.any.number.of.sub.properties}}}
```

where *indexOfOnArrayElement* is an integer and the following properties are dependent on the contents of the matched object. When template output will appear on the screen (e.g., in a visual report), use two curly braces ({{ }}) instead of three ({{{ }}}) to properly escape HTML characters.

Note: For the curious, the dim-boilerplate uses the mustache templating library to render templates. Any mustache syntax should work, but keep security in mind if you are crafting a game with private player data and user manipulable templates.

Default

The default object includes configuration information used by the *controllers* and *views* included in the boilerplate. Its keys and values are completely open-ended: you may add whatever fields you need to support your customizations to the boilerplate. There may be one and only one default object in the world array.

The following is the default object as shipped in the boilerplate:

```
{
    "type": "default",
    /* controller to use when scenes do not specify one */
    "controller": "dim/controllers/explore/explore",

    /* maps scene / item properties to output channels */
    "objectReport": {
        /* on user selection in a controller */
        "user.select": [
            {
                "visual.name": "title",
                "aural.name": "narration",
                "aural.sound": "sound"
            }
        ],
        /* on user activation in a controller */
        "user.activate": [
            {
                "visual.name": "title"
            }
        ]
    },

    /* defines channels for aural / visual rendering */
    "channels": {
        /* continuous, background audio channel */
        "ambience": {
            "type": "aural",
            "loop": True,
            /* does not stop if interrupted by the user, only if a sound with a␣
↪different URI attempts is queued on the channel */
            "swapstop": True,
            /* volume gain, 0.0 to 1.0 */
            "gain": 0.15,
            /* crossfade when changing sounds */
            "crossfade": True
        },
        /* channel for brief sound effects */
        "sound": {
            "type": "aural",
            "gain": 0.7
        },
        /* channel for spoken narration */
```

```
        "narration": {
            "type": "aural",
            "gain": 0.9
        },
        /* channel for visual backgrounds */
        "backdrop": {
            "type": "visual"
        },
        /* channel for lengthy text descriptions */
        "description": {
            "type": "visual"
        },
        /* channel for brief text descriptions */
        "title": {
            "type": "visual"
        }
    }
}
```

Note: You can define additional aural and visual channels in the defaults. New aural channels will output speech and sound if used in reports. New visual channels, however, require changes to the *visual view* to place their output on screen.

Controller Resources

The final type of object supported in the world array holds information for use by controllers. These objects with type *ctrl* may contain arbitrary properties: whatever the controller needs to function. The world array may contain zero or more controller objects.

For example, a game event might activate an end-of-game controller when the player wins or loses. The event might pass the ID of a *ctrl* object which the controller might reference in making its report of the player's win or loss.

```
{
    "type": "ctrl",
    "id": "win",
    "report": [
        {
            "title": "You Win"
        }
    ]
}
```

Media Assets

Objects in the world array may reference external, non-text media assets such as sounds and images. Many of the examples above include media URIs which fit this format:

```
[type]://[path/relative/to/world.json]
```

Sounds

Audio file URIs have the type *sound* and include a path and filename without a filename extension. The audio view attaches an extension of ".ogg" or ".mp3" depending on which audio codec the player's browser supports. You should package both OGG Vorbis and MP3 versions of all of your audio files with your game to ensure cross browser compatibility.

Some examples of valid audio URIs include:

```
sound://sound_effects/sizzling
sound://narration/cookSinging
sound://audio_files/music/track1
```

Images

Image file URIs have the type *image* and include a path and file with a filename extension. Numerous image formats are ubiquitously supported across browsers (e.g., PNG, JPEG, GIF), so there is no need to detect browser capabilities or ship images in multiple formats.

Some examples of valid image URIs include:

```
image://backdrops/ocean.jpg
image://sprites/egg.png
```

Stylesheets

You can modify the basic styling of the boilerplate by adjusting CSS rules without touching the view JavaScript or HTML. Open the *webapp/css/main.css* file and make your edits in the section with the comment header *Author's custom styles*. You can easily change the font family, font size, foreground color, background color, and so on in here. For more advanced changes to the visuals in the boilerplate UI, see *Level 3: Extend the Engine*.

CHAPTER 3

Level 2: Code New Controllers

Lifecyle

Resources

Input

Devices

Styles

CHAPTER 4

Level 3: Extend the Engine

MVC Breakdown

Model

View

Controller

Pub-Sub

Topics

CHAPTER 5

Wrap Up

Deploy Your Game

Contribute to the Boilerplate

Index