TextWorld Documentation

Release 0.1-beta

Marc-Alexandre Côté, Tavian Barnes, Matthew Hausknecht, James

Feb 08, 2019

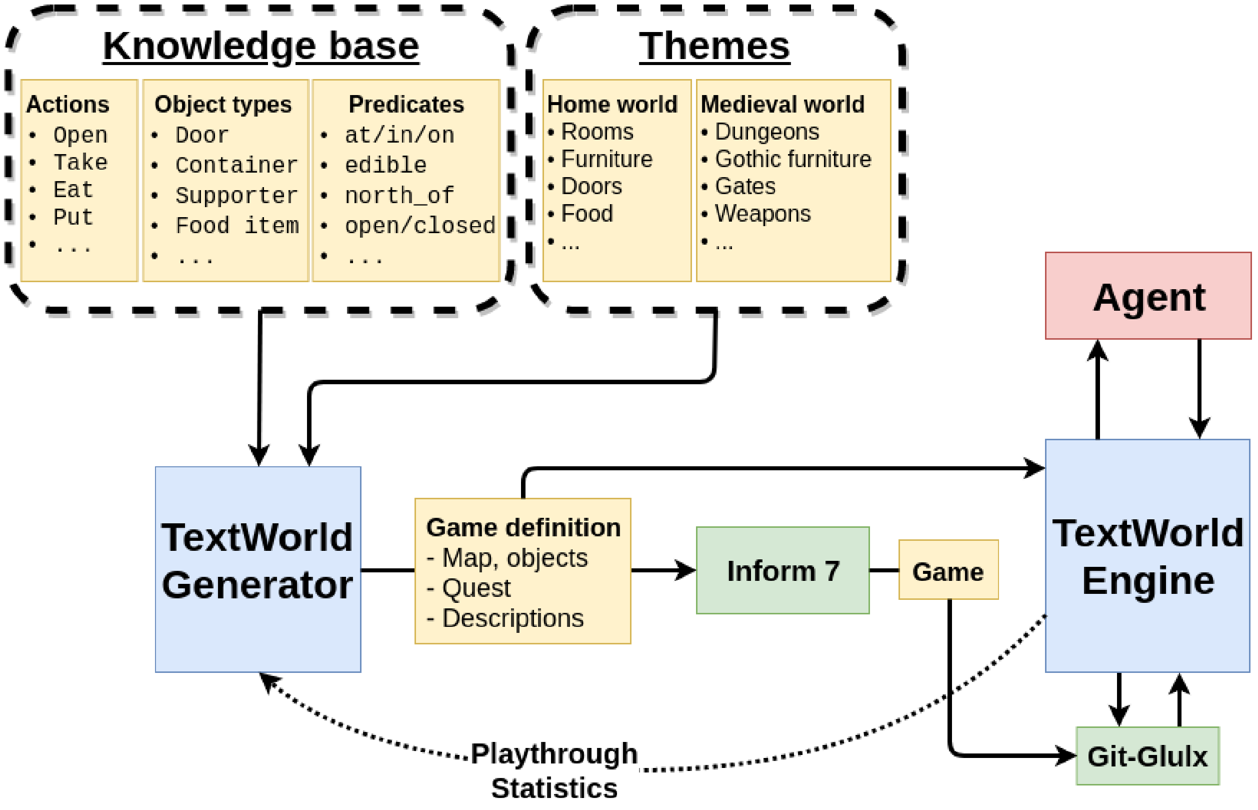
Notes

TextWorld is a text-based learning environment for Reinforcement Learning agent.

CHAPTER 1

What is TextWorld?

TextWorld is a sandbox learning environment for training and testing reinforcement learning (RL) agents on text-based games. It enables generating games from a game distribution parameterized by the map size, the number of objects, quest length and complexity, richness of text descriptions, and more. Then, one can sample game from that distribution. TextWorld can also be used to play existing text-based games.



CHAPTER 2

Custom Environment

Making new Environments is useful when games output more information than needed (e.g. a header). It is also useful for detecting when a game ends (either win or lose) since most games have different ways of letting the player knows the game has ended.

This page will guide you through creating a new environment. You should have prior knowledge of the `textworld.core.Environment` and `textworld.core.GameState` classes.

2.1 Zork1Environment

For this tutorial, we will be creating an Environment dedicated to Zork1. You can get a copy of the game from the Internet Archive).

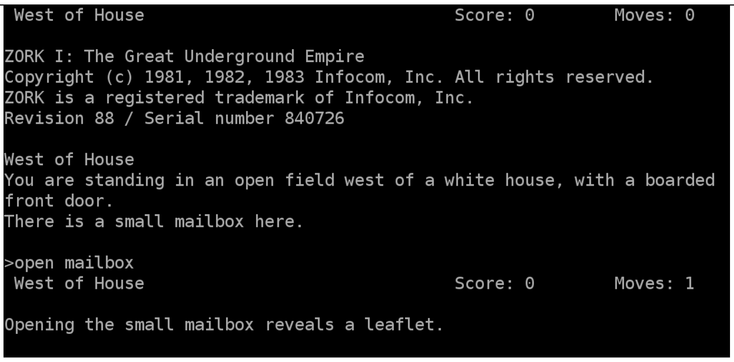
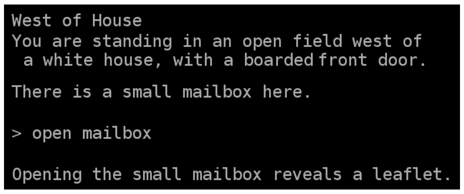
2.1.1 Why do we want a new environment?

Let's start by running the game directly using the native frotz interpreter as follows

```
dfrotz path/to/zork1.z5
```

As we can see (see left column in the table below), Zork1 prints out a header displaying the name of the room, the scores and the number of moves. Also, when we start the game, some version numbers and other information are printed.

All this decorative text is just noise really. One could argue that header contains meaningful information but the `game_state` object the agent receives already should have all that information. So, there is not point in making the task more difficult for the agent than it already is.

Frotz	TextWorld
	

```
 West of House                                   Score: 0        Moves: 0

ZORK I: The Great Underground Empire
Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights reserved.
ZORK is a registered trademark of Infocom, Inc.
Revision 88 / Serial number 840726

West of House
You are standing in an open field west of a white house, with a boarded
front door.
There is a small mailbox here.

>open mailbox
 West of House                                   Score: 0        Moves: 1

Opening the small mailbox reveals a leaflet.
```

```
West of House
You are standing in an open field west of
 a white house, with a boarded front door.
There is a small mailbox here.

> open mailbox

Opening the small mailbox reveals a leaflet.
```

As we can see in the right column of the above table, the input text is more compact.

2.1.2 Building the Zork1Environment

Since `zork1.z5` is a Z-Machine game, we want to leverage the *python-frotz* communication pipeline already in place in `textworld.envs.FrotzEnvironment`. To do so, we simply have to subclass it.

```python
from textworld.envs import FrotzEnvironment

class Zork1Environment(FrotzEnvironment):
    GAME_STATE_CLASS = Zork1GameState
```

As we can see there is nothing much to modify for this particular environment other than specifying the game state class to use. This is because we only need to clean the output text rather than changing some fundamental behavior of the `FrotzEnvironment`.

Cleaning up the output is essentially done in the `Zork1GameState` (a subclass of `GameState`) using a bunch of regular expressions. Here's what it looks like

```python
import textworld

class Zork1GameState(textworld.GameState):

    def _remove_header(self, text):
        cleaned_text = text_utils.remove_header(text)
        return cleaned_text.lstrip("\n")

    def _check_for_death(self, text):
        return "****  You have died  ****" in text

    @property
    def nb_deaths(self):
        """ Number of times the player has died. """
        if not hasattr(self, "_nb_deaths"):
            if self.previous_state is None:
                self._nb_deaths = 0
            else:
                has_died = self._check_for_death(self.feedback)
                self._nb_deaths = self.previous_state.nb_deaths + has_died

        return self._nb_deaths
```

(continues on next page)

(continued from previous page)

```
    @property
    def feedback(self):
        """ Interpreter's response after issuing last command. """
        if not hasattr(self, "_feedback"):
            # Extract feeback from command's output.
            self._feedback = self._remove_header(self._raw)
            if self.previous_state is None:
                # Remove version number and copyright text.
                self._feedback = "\n".join(self._feedback.split("\n")[5:])

        return self._feedback

    @property
    def inventory(self):
        """ Player's inventory. """
        if not hasattr(self, "_inventory"):
            # Issue the "inventory" command and parse its output.
            text = self._env.send("inventory")
            self._inventory = self._remove_header(text)

        return self._inventory

    def _retrieve_score(self):
        if self.has_won or self.has_lost:
            _score_text = self.feedback
        else:
            # Issue the "score" command and parse its output.
            text = self._env.send("score")
            _score_text = self._remove_header(text)

        regex = r"Your score is (?P<score>[0-9]+) \(total of (?P<max_score>[0-9]+)␣
↪points\)"
        match = re.match(regex, _score_text)
        self._score = int(match.groupdict()['score'].strip())
        self._max_score = int(match.groupdict()['max_score'].strip())
        return self._score, self._max_score

    @property
    def score(self):
        """ Current score. """
        if not hasattr(self, "_score"):
            self._retrieve_score()

        return self._score

    @property
    def max_score(self):
        """ Max score for this game. """
        if not hasattr(self, "_max_score"):
            self._retrieve_score()

        return self._max_score

    @property
    def description(self):
        """ Description of the current location. """
```

(continues on next page)

(continued from previous page)

```
        if not hasattr(self, "_description"):
            # Issue the "look" command and parse its output.
            text = self._env.send("look")
            self._description = self._remove_header(text)

        return self._description

    @property
    def has_won(self):
        """ Whether the player has won the game or not. """
        return "Inside the Barrow" in self.feedback.split("\n")[0]

    @property
    def has_lost(self):
        """ Whether the player has lost the game or not. """
        return self.nb_deaths >= 3
```

Then the last thing to do is to make TextWorld framework aware of that new environment. This is done by adding a new entry to the `CUSTOM_ENVIRONMENTS` dictionary located in `textworld/envs/__init__.py`.

```
# Import dedicated environment
from textworld.envs.frotz.zork1 import Zork1Environment

CUSTOM_ENVIRONMENTS = {
    "zork1.z5": Zork1Environment
}
```

With everything in place, we can check the results using `tw-play zork.z5`.

CHAPTER 3

Known Issues

3.1 Inform 7

Inform 7 command line tools don't support Windows Linux Subsystem (a.k.a Bash on Ubuntu on Windows).

3.2 FrotzEnvironment

This is known to cause some concurrency issues when commands are rapidely sent to the game's interpreter. An alternative is to use the `JerichoEnvironment` for Z-Machine games or `JerichoEnvironment` for games generated with TextWorld.

CHAPTER 4

textworld

4.1 Core

CHAPTER 5

textworld.gym

5.1 Agent

5.2 Envs

5.3 Spaces

CHAPTER 6

textworld.envs

6.1 Glulx

6.2 Wrappers

6.3 Z-Machine

CHAPTER 7

textworld.agents

CHAPTER 8

textworld.generator

8.1 Game

8.2 World

8.3 GameMaker

8.4 Grammar

8.5 Knowledge Base

8.5.1 Data

container.twl

```
# container
type c : t {
    predicates {
        open(c);
        closed(c);
        locked(c);

        in(o, c);
    }

    rules {
        lock/c   :: $at(P, r) & $at(c, r) & $in(k, I) & $match(k, c) & closed(c) ->␣
↪locked(c);
        unlock/c :: $at(P, r) & $at(c, r) & $in(k, I) & $match(k, c) & locked(c) ->␣
↪closed(c);
```

(continues on next page)

(continued from previous page)

```
        open/c  :: $at(P, r) & $at(c, r) & closed(c) -> open(c);
        close/c :: $at(P, r) & $at(c, r) & open(c) -> closed(c);
    }

    reverse_rules {
        lock/c :: unlock/c;
        open/c :: close/c;
    }

    constraints {
        c1 :: open(c)   & closed(c) -> fail();
        c2 :: open(c)   & locked(c) -> fail();
        c3 :: closed(c) & locked(c) -> fail();
    }

    inform7 {
        type {
            kind :: "container";
            definition :: "containers are openable, lockable and fixed in place.␣
↪containers are usually closed.";
        }

        predicates {
            open(c) :: "The {c} is open";
            closed(c) :: "The {c} is closed";
            locked(c) :: "The {c} is locked";

            in(o, c) :: "The {o} is in the {c}";
        }

        commands {
            open/c :: "open {c}" :: "opening the {c}";
            close/c :: "close {c}" :: "closing the {c}";

            lock/c :: "lock {c} with {k}" :: "locking the {c} with the {k}";
            unlock/c :: "unlock {c} with {k}" :: "unlocking the {c} with the {k}";
        }
    }
}
```

door.twl

```
# door
type d : t {
    predicates {
        open(d);
        closed(d);
        locked(d);

        link(r, d, r);
    }

    rules {
        lock/d   :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & $in(k, I) &
↪$match(k, d) & closed(d) -> locked(d);
```

(continues on next page)

(continued from previous page)

```
        unlock/d :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & $in(k, I) &
↪$match(k, d) & locked(d) -> closed(d);

        open/d   :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & closed(d) ->␣
↪open(d) & free(r, r') & free(r', r);
        close/d  :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & open(d) & free(r,␣
↪r') & free(r', r) -> closed(d);

        examine/d :: at(P, r) & $link(r, d, r') -> at(P, r);  # Nothing changes.
    }

    reverse_rules {
        lock/d :: unlock/d;
        open/d :: close/d;
    }

    constraints {
        d1 :: open(d)   & closed(d) -> fail();
        d2 :: open(d)   & locked(d) -> fail();
        d3 :: closed(d) & locked(d) -> fail();

        # A door can't be used to link more than two rooms.
        link1 :: link(r, d, r') & link(r, d, r'') -> fail();
        link2 :: link(r, d, r') & link(r'', d, r''') -> fail();

        # There's already a door linking two rooms.
        link3 :: link(r, d, r') & link(r, d', r') -> fail();

        # There cannot be more than four doors in a room.
        too_many_doors :: link(r, d1: d, r1: r) & link(r, d2: d, r2: r) & link(r, d3:␣
↪d, r3: r) & link(r, d4: d, r4: r) & link(r, d5: d, r5: r) -> fail();

        # There cannot be more than four doors in a room.
        dr1 :: free(r, r1: r) & link(r, d2: d, r2: r) & link(r, d3: d, r3: r) &␣
↪link(r, d4: d, r4: r) & link(r, d5: d, r5: r) -> fail();
        dr2 :: free(r, r1: r) & free(r, r2: r) & link(r, d3: d, r3: r) & link(r, d4:␣
↪d, r4: r) & link(r, d5: d, r5: r) -> fail();
        dr3 :: free(r, r1: r) & free(r, r2: r) & free(r, r3: r) & link(r, d4: d, r4:␣
↪r) & link(r, d5: d, r5: r) -> fail();
        dr4 :: free(r, r1: r) & free(r, r2: r) & free(r, r3: r) & free(r, r4: r) &␣
↪link(r, d5: d, r5: r) -> fail();

        free1 :: link(r, d, r') & free(r, r') & closed(d) -> fail();
        free2 :: link(r, d, r') & free(r, r') & locked(d) -> fail();
    }

    inform7 {
        type {
            kind :: "door";
            definition :: "door is openable and lockable.";
        }

        predicates {
            open(d) :: "The {d} is open";
            closed(d) :: "The {d} is closed";
            locked(d) :: "The {d} is locked";
            link(r, d, r') :: "";  # No equivalent in Inform7.
```

(continues on next page)

(continued from previous page)

```
        }

        commands {
            open/d :: "open {d}" :: "opening {d}";
            close/d :: "close {d}" :: "closing {d}";

            unlock/d :: "unlock {d} with {k}" :: "unlocking {d} with the {k}";
            lock/d :: "lock {d} with {k}" :: "locking {d} with the {k}";

            examine/d :: "examine {d}" :: "examining {d}";
        }
    }
}
```

food.twl

```
# food
type f : o {
    predicates {
        edible(f);
        eaten(f);
    }

    rules {
        eat :: in(f, I) -> eaten(f);
    }

    constraints {
        eaten1 :: eaten(f) & in(f, I) -> fail();
        eaten2 :: eaten(f) & in(f, c) -> fail();
        eaten3 :: eaten(f) & on(f, s) -> fail();
        eaten4 :: eaten(f) & at(f, r) -> fail();
    }

    inform7 {
        type {
            kind :: "food";
            definition :: "food is edible.";
        }

        predicates {
            edible(f) :: "The {f} is edible";
            eaten(f) :: "The {f} is nowhere";
        }

        commands {
            eat :: "eat {f}" :: "eating the {f}";
        }
    }
}
```

inventory.twl

```
# Inventory
type I {
    predicates {
        in(o, I);
    }

    rules {
        inventory :: at(P, r) -> at(P, r);  # Nothing changes.

        take :: $at(P, r) & at(o, r) -> in(o, I);
        drop :: $at(P, r) & in(o, I) -> at(o, r);

        take/c :: $at(P, r) & $at(c, r) & $open(c) & in(o, c) -> in(o, I);
        insert :: $at(P, r) & $at(c, r) & $open(c) & in(o, I) -> in(o, c);

        take/s :: $at(P, r) & $at(s, r) & on(o, s) -> in(o, I);
        put    :: $at(P, r) & $at(s, r) & in(o, I) -> on(o, s);

        examine/I :: in(o, I) -> in(o, I);  # Nothing changes.
        examine/s :: at(P, r) & $at(s, r) & $on(o, s) -> at(P, r);  # Nothing changes.
        examine/c :: at(P, r) & $at(c, r) & $open(c) & $in(o, c) -> at(P, r);  #␣
↪Nothing changes.
    }

    reverse_rules {
        take :: drop;
        take/c :: insert;
        take/s :: put;
    }

    inform7 {
        predicates {
            in(o, I) :: "The player carries the {o}";
        }

        commands {
            take :: "take {o}" :: "taking the {o}";
            drop :: "drop {o}" :: "dropping the {o}";

            take/c :: "take {o} from {c}" :: "removing the {o} from the {c}";
            insert :: "insert {o} into {c}" :: "inserting the {o} into the {c}";

            take/s :: "take {o} from {s}" :: "removing the {o} from the {s}";
            put :: "put {o} on {s}" :: "putting the {o} on the {s}";

            inventory :: "inventory" :: "taking inventory";

            examine/I :: "examine {o}" :: "examining the {o}";
            examine/s :: "examine {o}" :: "examining the {o}";
            examine/c :: "examine {o}" :: "examining the {o}";
        }
    }
}
```

key.twl

```
# key
type k : o {
    predicates {
        match(k, c);
        match(k, d);
    }

    constraints {
        k1 :: match(k, c) & match(k', c) -> fail();
        k2 :: match(k, c) & match(k, c') -> fail();
        k3 :: match(k, d) & match(k', d) -> fail();
        k4 :: match(k, d) & match(k, d') -> fail();
    }

    inform7 {
        type {
            kind :: "key";
        }

        predicates {
            match(k, c) :: "The matching key of the {c} is the {k}";
            match(k, d) :: "The matching key of the {d} is the {k}";
        }
    }
}
```

object.twl

```
# object
type o : t {
    constraints {
        obj1 :: in(o, I) & in(o, c) -> fail();
        obj2 :: in(o, I) & on(o, s) -> fail();
        obj3 :: in(o, I) & at(o, r) -> fail();
        obj4 :: in(o, c) & on(o, s) -> fail();
        obj5 :: in(o, c) & at(o, r) -> fail();
        obj6 :: on(o, s) & at(o, r) -> fail();
        obj7 :: at(o, r) & at(o, r') -> fail();
        obj8 :: in(o, c) & in(o, c') -> fail();
        obj9 :: on(o, s) & on(o, s') -> fail();
    }

    inform7 {
        type {
            kind :: "object-like";
            definition :: "object-like is portable.";
        }
    }
}
```

player.twl

```
# Player
type P {
    rules {
        look :: at(P, r) -> at(P, r);  # Nothing changes.
    }

    inform7 {
        commands {
            look :: "look" :: "looking";
        }
    }
}
```

room.twl

```
# room
type r {
    predicates {
        at(P, r);
        at(t, r);

        north_of(r, r);
        west_of(r, r);

        north_of/d(r, d, r);
        west_of/d(r, d, r);

        free(r, r);

        south_of(r, r') = north_of(r', r);
        east_of(r, r') = west_of(r', r);

        south_of/d(r, d, r') = north_of/d(r', d, r);
        east_of/d(r, d, r') = west_of/d(r', d, r);
    }

    rules {
        go/north :: at(P, r) & $north_of(r', r) & $free(r, r') & $free(r', r) -> at(P,
↪ r');
        go/south :: at(P, r) & $south_of(r', r) & $free(r, r') & $free(r', r) -> at(P,
↪ r');
        go/east  :: at(P, r) & $east_of(r', r) & $free(r, r') & $free(r', r) -> at(P,␣
↪r');
        go/west  :: at(P, r) & $west_of(r', r) & $free(r, r') & $free(r', r) -> at(P,␣
↪r');
    }

    reverse_rules {
        go/north :: go/south;
        go/west :: go/east;
    }

    constraints {
        r1 :: at(P, r) & at(P, r') -> fail();
```

(continues on next page)

(continued from previous page)

```
        r2 :: at(s, r) & at(s, r') -> fail();
        r3 :: at(c, r) & at(c, r') -> fail();

        # An exit direction can only lead to one room.
        nav_rr1 :: north_of(r, r') & north_of(r'', r') -> fail();
        nav_rr2 :: south_of(r, r') & south_of(r'', r') -> fail();
        nav_rr3 :: east_of(r, r') & east_of(r'', r') -> fail();
        nav_rr4 :: west_of(r, r') & west_of(r'', r') -> fail();

        # Two rooms can only be connected once with each other.
        nav_rrA :: north_of(r, r') & south_of(r, r') -> fail();
        nav_rrB :: north_of(r, r') & west_of(r, r') -> fail();
        nav_rrC :: north_of(r, r') & east_of(r, r') -> fail();
        nav_rrD :: south_of(r, r') & west_of(r, r') -> fail();
        nav_rrE :: south_of(r, r') & east_of(r, r') -> fail();
        nav_rrF :: west_of(r, r')  & east_of(r, r') -> fail();
    }

    inform7 {
        type {
            kind :: "room";
        }

        predicates {
            at(P, r) :: "The player is in {r}";
            at(t, r) :: "The {t} is in {r}";
            free(r, r') :: "";  # No equivalent in Inform7.

            north_of(r, r') :: "The {r} is mapped north of {r'}";
            south_of(r, r') :: "The {r} is mapped south of {r'}";
            east_of(r, r') :: "The {r} is mapped east of {r'}";
            west_of(r, r') :: "The {r} is mapped west of {r'}";

            north_of/d(r, d, r') :: "South of {r} and north of {r'} is a door called
↪{d}";
            south_of/d(r, d, r') :: "North of {r} and south of {r'} is a door called
↪{d}";
            east_of/d(r, d, r') :: "West of {r} and east of {r'} is a door called {d}
↪";
            west_of/d(r, d, r') :: "East of {r} and west of {r'} is a door called {d}
↪";
        }

        commands {
            go/north :: "go north" :: "going north";
            go/south :: "go south" :: "going south";
            go/east :: "go east" :: "going east";
            go/west :: "go west" :: "going west";
        }
    }
}
```

supporter.twl

```
# supporter
type s : t {
    predicates {
        on(o, s);
    }

    inform7 {
        type {
            kind :: "supporter";
            definition :: "supporters are fixed in place.";
        }

        predicates {
            on(o, s) :: "The {o} is on the {s}";
        }
    }
}
```

thing.twl

```
# thing
type t {
    rules {
        examine/t :: at(P, r) & $at(t, r) -> at(P, r);
    }

    inform7 {
        type {
            kind :: "thing";
        }

        commands {
            examine/t :: "examine {t}" :: "examining the {t}";
        }
    }
}
```

8.6 Inform 7

CHAPTER 9

textworld.challenges

CHAPTER 10

textworld.logic

CHAPTER 11

textworld.render

CHAPTER 12

textworld.utils

CHAPTER 13

Indices and tables

- genindex
- modindex
- search