

Lecture #17. 게임 데이터

2D 게임 프로그래밍

이대현 교수



한국공학대학교
TECH UNIVERSITY OF KOREA

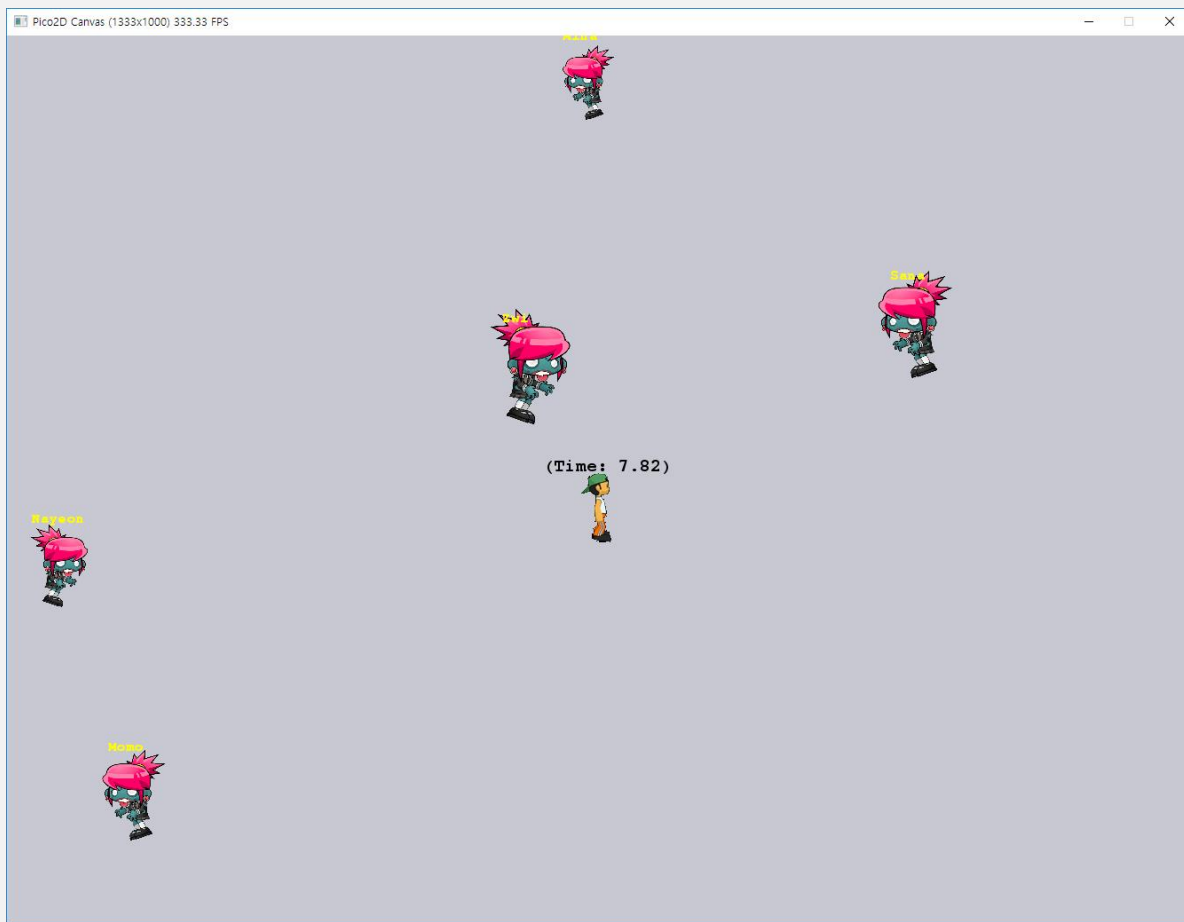
학습 내용

- 직렬화
- 객체들의 초기값 설정
- 게임 Save와 Load의 구현

직렬화(Serialization)

- 프로그램 내의 객체 데이터를 외부에 저장 또는 보내는 행위.
- 나중에 다시 복구(de-serialization)할 수 있어야 함.
- 직렬화의 활용
 - 게임 플레이 상황을 저장(Save)하고 로드(Load).
 - 맵 데이터의 출력(Export) 및 입력(Import)
 - 게임내 객체들의 초기화 데이터 로딩
 - 게임 결과 및 기록 저장

객체들의 초기 상태 정의를 어떻게 할까?



흔히 하는 “무식한 방법”: 하드코딩(Hard Coding)

```
zombie = Zombie()  
zombie.x = 4           # 4 meters  
zombie.y = 3           # 3 meters  
zombie.size = 1.5
```

문제점은?
왜 무식한가?

소프트 코딩(Soft Coding)

x : 4
y : 3
size : 1.5



```
data_struct = {"total_spam":0,"total_ham":0,"total_spam_words":0,"total_ham_words":0,"bag_of_words":{}}

def learn(message, messagetype):
    bag_of_words = data_struct["bag_of_words"]
    words = message.split(" ")
    for word in words:
        try:
            bag_of_words[word][messagetype] += 1
        except KeyError:
            bag_of_words[word] = [1-messagetype, messagetype]
            data_struct["total_spam_words"] += messagetype
            data_struct["total_ham_words"] += (1-messagetype)

    data_struct["total_spam"] += messagetype
    data_struct["total_ham"] += (1-messagetype)
#K is laplacian smoother
def predict(message, K):
    bag_of_words = data_struct["bag_of_words"]
    k=K
    #total messages
    total_messages = data_struct["total_spam"] + data_struct["total_ham"]
    #prior probability of spam
    p_s = (data_struct["total_spam"] + k) / (total_messages * 1.0 * k**2)
    #prior probability of ham
    p_h = (data_struct["total_ham"] + k) / (total_messages * 1.0 * k**2)
    words = message.split(" ")
    #p_m_s of message given its spam && p_m_h probability of message given its ham
    p_m_s = 1
    p_m_h = 1
    for word in words:
        p_m_s *= (bag_of_words[word][1] * 1.0 * k) / (data_struct["total_spam_words"] + k * (len(bag_of_words)))
        p_m_h *= (bag_of_words[word][0] * 1.0 * k) / (data_struct["total_ham_words"] + k * (len(bag_of_words)))
    #bayes rule && p_s_m probability of spam given a particular message
    p_s_m = p_s * p_m_s / (p_m_s * p_s + p_m_h * p_h)
    return p_s_m

#1 corresponds to message is spam and 0 that it is ham
learn("offer is secret", 1)
learn("click secret link", 1)
learn("secret sports link", 1)
learn("play sports today", 0)
learn("went play sports", 0)
learn("secret sports event", 0)
learn("sports is today", 0)
learn("sports cost money", 0)

print predict("today is secret", 1)
```

파이썬 파일 입출력

- C와 매우 유사
- `open()`으로 열고, `close()`로 닫음.
- `write()`, `read()`는 각각 `str` 을 쓰고 읽을 수 있음.(text mode 일 경우)

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

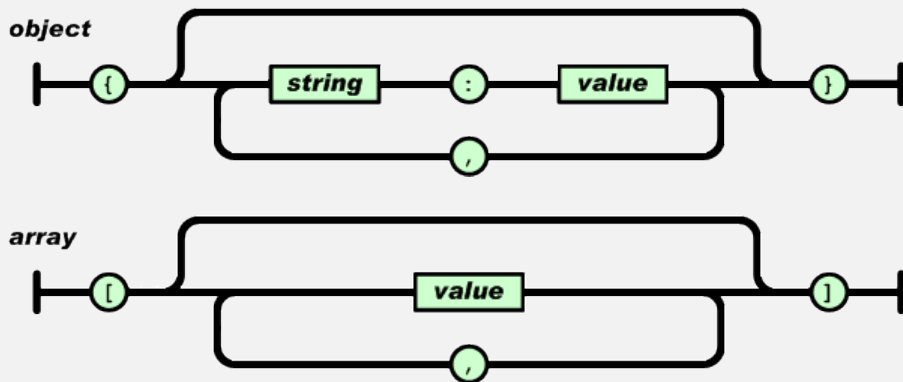
문자	의미
'r'	읽기용으로 엽니다 (기본값)
'w'	쓰기용으로 엽니다, 파일을 먼저 자릅니다.
'x'	독점적인 파일 만들기용으로 엽니다, 이미 존재하는 경우에는 실패합니다.
'a'	쓰기용으로 엽니다, 파일이 존재하는 경우는 파일의 끝에 덧붙입니다
'b'	바이너리 모드
't'	텍스트 모드 (기본값)
'+'	갱신(읽기 및 쓰기)용으로 디스크 파일을 엽니다
'U'	유니버설 줄 넘김 모드 (디프리케이트 되었습니다)

```
data = {'x': 10, 'y':20, 'size':1.5}  
file = open('data.txt', 'w')  
file.write(str(data))  
file.close()
```

```
file = open('data.txt', 'r')  
data_str = file.read()  
file.close()  
print(data_str)
```


JSON(Java Script Object Notation)

- File write/read 의 경우, 문자열을 실제 데이터로 변환하는 작업이 필수적임. -> 까다롭고 복잡하고, 에러 발생의 소지가 많음.
- 객체를 교환(저장 및 전송 등)하기 위한 텍스트 형식 표준
- 파이썬의 리스트와 딕셔너리와 거의 동일



zombie_data.json

```
[  
  {"name": "Nayeon", "x": 2, "y": 2, "size": 1.0},  
  {"name": "Momo", "x": 8, "y": 3, "size": 1.1},  
  {"name": "Sana", "x": 30, "y": 20, "size": 1.3},  
  {"name": "Mina", "x": 10, "y": 30, "size": 0.9},  
  {"name": "Zwi", "x": 20, "y": 40, "size": 1.4}  
]
```

json 형식에서는 문자열
표시를 큰따옴표만 허용!

Python JSON Module의 마법

```
import json
```

```
s = json.dumps(o)    # 객체 obj 를 문자열 s로 변환
```

```
o = json.loads(s)    # 문자열 s 를 객체 o로 변환
```

```
json.dump(o, f)      # 객체 o를 파일 f로 저장
```

```
o = json.load(f)     # 파일 f를 로드해서 객체 o로 변환
```



```
import json

numbers = [1,2,3,4]
numbers_string = json.dumps(numbers)
print(numbers_string)

values_string = '{"x":10, "y":20, "size":4.5}'
values = json.loads(values_string)
print(values)
```



JSON를 활용한 객체 초기화

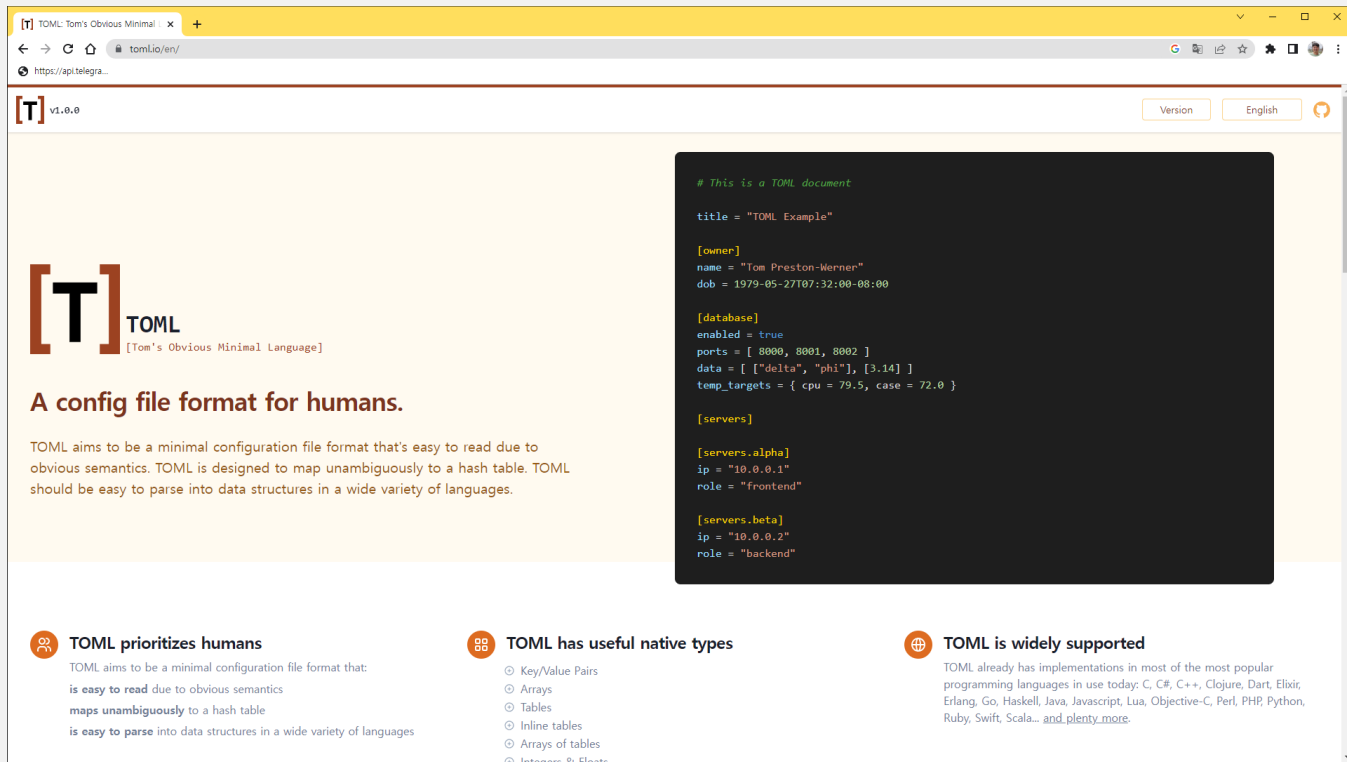


```
def create_new_world():
    server.boy = Boy()
    game_world.add_object(server.boy, 1)
    game_world.add_collision_pairs(server.boy, None, 'boy:zombie')

    # load json object data
    with open('zombie_data.json', 'r') as f:
        zombie_data_list = json.load(f)
        zombies = [Zombie(o['name'], o['x'], o['y'], o['size']) for o in zombie_data_list]
        game_world.add_objects(zombies, 1)
        game_world.add_collision_pairs(None, zombies, 'boy:zombie')
```

TOML

■ 설정 데이터를 좀 더 읽기 쉽게 저장하기 위한 텍스트 파일 형식



The screenshot shows the TOML website. On the left, there's a large TOML logo with the text "[TOML] TOML [Tom's Obvious Minimal Language]". Below it, the heading "A config file format for humans." is followed by a paragraph explaining that TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics and is designed to map unambiguously to a hash table. On the right, a dark-themed code block displays a sample TOML document. At the bottom, three key features are listed with icons: TOML prioritizes humans, TOML has useful native types, and TOML is widely supported.

[TOML] TOML
[Tom's Obvious Minimal Language]

A config file format for humans.

TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

```
# This is a TOML document

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [ 8000, 8001, 8002 ]
data = [ ["delta", "phi"], [3.14] ]
temp_targets = { cpu = 79.5, case = 72.0 }

[servers]

[servers.alpha]
ip = "10.0.0.1"
role = "frontend"

[servers.beta]
ip = "10.0.0.2"
role = "backend"
```

TOML prioritizes humans
TOML aims to be a minimal configuration file format that:
is **easy to read** due to obvious semantics
maps **unambiguously** to a hash table
is **easy to parse** into data structures in a wide variety of languages

TOML has useful native types

- Key/Value Pairs
- Arrays
- Tables
- Inline tables
- Arrays of tables
- Integer & Float

TOML is widely supported
TOML already has implementations in most of the most popular programming languages in use today: C, C#, C++, Clojure, Dart, Elixir, Erlang, Go, Haskell, Java, Javascript, Lua, Objective-C, Perl, PHP, Python, Ruby, Swift, Scala... and plenty more.

```
# This is a TOML document

title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
dob = 1979-05-27T07:32:00-08:00

[database]
enabled = true
ports = [ 8000, 8001, 8002 ]
data = [ [ "delta", "phi" ], [ 3.14 ] ]
temp_targets = { cpu = 79.5, case = 72.0 }

[servers]

[servers.alpha]
ip = "10.0.0.1"
role = "frontend"

[servers.beta]
ip = "10.0.0.2"
role = "backend"
```


TOML 를 이용한 초기 데이터 로드

```
# load toml object data
with open('zombie_data.toml', 'rb') as f:
    zombie_list = tomllib.load(f)['zombie']['objects']
    for o in zombie_list:
        zombie = Zombie(o['name'], o['x'], o['y'], o['size'])
        game_world.add_object(zombie, 1)
```

게임 Save 와 Load

- 게임 플레이 중, 특정 상황을 저장하고, 나중에 다시 불러와서 쓰는 기능.
- 대부분의 single play 게임들은 이 기능을 갖고 있음.
- 가장 쉽게 구현하는 방법은 게임 월드의 모든 객체의 상태를 저장하고, 나중에 불러오는 것.
- JSON 이용?
 - 간단히 객체들의 저장은 되지만, JSON의 가장 기본적인 data type 만을 지원하기 때문에, 게임 내 객체들의 다양한 데이터를 저장하는 것은 번거로울 수 있음.
- 가장 확실한 해법은?
 - Python Pickle !!!!

Pickle(<https://docs.python.org/ko/3/library/pickle.html>)

- 파이썬이 제공하는 객체 직렬화 모듈
- 대부분의 파이썬 내부 데이터들을 직렬화할 수 있음.

어떤 것이 피클 되고 역 피클 될 수 있을까요?

다음 형을 피클 할 수 있습니다:

- `None`, `True` 와 `False`
- 정수, 실수, 복소수
- 문자열, 바이트열, 바이트 배열(`bytearray`)
- 피클 가능한 객체만 포함하는 튜플, 리스트, 집합과 딕셔너리
- 모듈의 최상위 수준에서 정의된 함수 (`lambda` 가 아니라 `def` 를 사용하는)
- 모듈의 최상위 수준에서 정의된 내장 함수
- 모듈의 최상위 수준에서 정의된 클래스
- 그런 클래스의 인스턴스 중에서 `__dict__` 나 `__getstate__()` 를 호출한 결과가 피클 가능한 것들 (자세한 내용은 클래스 인스턴스 피클링 절을 참조하세요).



Pickle 모듈 사용 예 (1)

```
import pickle
```

```
data = [1,2,3,4,5]
```

```
with open('data.pickle', 'wb') as f:  
    pickle.dump(data, f)
```

```
with open('data.pickle', 'rb') as f:  
    read_data = pickle.load(f)
```

```
print(read_data)
```

list 객체를 피클하고,
역피클할 수 있음.

Pickle 모듈 사용 예 (2)

사용자 정의 객체를 피클, 역피클

```
class Npc:
    def __init__(self, name, x, y):
        self.name = name
        self.x, self.y = x, y

yuri = Npc('Yuri', 100, 200)

with open('npc.pickle', 'wb') as f:
    pickle.dump(yuri, f)

with open('npc.pickle', 'rb') as f:
    read_npc = pickle.load(f)

print(read_npc.name, read_npc.x, read_npc.y)
```

Pickle 로 안 되는 것들..

- 클래스 변수
- 순수 파이썬이 아닌, 외부 라이브러리를 통해서 획득한 데이터

File "W:\WorkCodingLive\2022-2DGP-Master\Labs\Lecture17_Game_Data\game_world.py", line 89, in save
pickle.dump(objects, f)
ValueError: ctypes objects containing pointers cannot be pickled

Dictionary 의 업데이트

```
data = {"name": "Yuri", "x":100, "y":200}

data['name'] = 'Yuna'

print(data)

new_data = {"name": "Jisu", "x":400, "y":900}

data.update(new_data)

print(data)
```

__dict__

- 클래스를 이용해서 생성한 객체는 모든 속성(멤버 변수)을 dictionary 형태로 내부적으로 저장하여 사용함.
- obj.__dict__ 라는 내부변수가 바로 그것임.
- 이것을 통해, 객체의 속성을 쉽게 바꿀 수 있음. - 다른 dictionary 데이터를 이용해서.

```
class NPC:
    def __init__(self, name, x, y):
        self.name = name
        self.x, self.y = x, y

yuri = NPC('Yuri', 100, 200)

print(yuri.__dict__)

new_data = {"name": "Jusu", "x": 400, "y": 900}

yuri.__dict__.update(new_data)

print(yuri.__dict__)
print(yuri.name, yuri.x, yuri.y)
```




Pickle을 활용한 게임 저장



```
def save():  
    game = [objects, collision_group]  
    with open('game.sav', 'wb') as f:  
        pickle.dump(game, f)  
  
def load():  
    global objects, collision_group  
    with open('game.sav', 'rb') as f:  
        game = pickle.load(f)  
        objects, collision_group = game[0], game[1]
```



```
def load_saved_world():  
    game_world.load()  
    for o in game_world.all_objects():  
        if isinstance(o, Boy):  
            server.boy = o  
            break
```



```
class Boy:
```

```
    def __getstate__(self):
        state = {'x': self.x, 'y': self.y, 'dir': self.dir,
                 'cur_state': self.cur_state}
        return state

    def __setstate__(self, state):
        self.__init__()
        self.__dict__.update(state)
```



```
class Zombie:
```

```
    def __getstate__(self):
        state = {'x': self.x, 'y': self.y, 'dir': self.dir,
                 'name': self.name, 'size': self.size}
        return state

    def __setstate__(self, state):
        self.__init__()
        self.__dict__.update(state)
```

게임월드의 저장

- world를 통째로 피클링하면 됨.

```
def save():  
    game = [objects, collision_group]  
    with open('game.sav', 'wb') as f:  
        pickle.dump(game, f)  
  
def load():  
    global objects, collision_group  
    with open('game.sav', 'rb') as f:  
        game = pickle.load(f)  
        objects, collision_group = game[0], game[1]
```

```
def load_saved_world():  
  
    game_world.load()  
    for o in game_world.all_objects():  
        if isinstance(o, Boy):  
            server.boy = o  
            break
```

객체 o 가 Boy 클래스로부터 생성된 객체인지
체크한 후, 맞으면, 이 객체를 boy 객체로 설정.

Boy 객체의 저장

- 객체의 멤버 변수들을 모두 저장할 필요가 없는 경우가 대부분.
- 저장이 필요한 내용만 선택적으로 골라서 저장할 수 있음.
- 피클링이 필요한 멤버 변수만 저장하고 복구함.

```
class Boy:
```

`__getstate__` : 객체를 저장할 때, 필요한 내용을 결정.
피클 모듈은 객체의 `__getstate__` 함수를 이용해서, 저장할 내용을 가져옴.

```
def __getstate__(self):  
    state = {'x': self.x, 'y': self.y, 'dir': self.dir,  
            'cur_state': self.cur_state}  
    return state
```

`__getstate__` : 객체를 복구할 때, 복구할 내용을 결정.

```
def __setstate__(self, state):
```

```
    self.__init__()
```

`pickle.load` 를 이용해서 객체를 만들 경우, `__init__()` 가 자동 호출되지 않으므로, 수동으로 호출함. 저장된 내용 이외의 값들을 채우기 위함.

```
    self.__dict__.update(state)
```