Содержание

1 Введение

Первоначально задача для практики заключалась в анализе поведения игроков в трехмерном шутере. В качестве такого шутера была выбрана игра «Counter-Strike: Global Offensive». Решающим фактором при выборе стала возможность записывать игру (т. н. демо-файлы, '`.dem`') и позже воспроизвести запись средствами самой игры.

В скоре после начала работы выяснилось, что сам игровой движок source инструментов для анализа не предоставляет. Более того, он даже не предоставляет возможности прямой работы с демо-файлами. Стало очевидно, что без подабающих инструментов и, самое главное, парсера демо-файлов, анализ выполнить не удастся. Отсутствие же какой-либо документации по формату '`.dem`' наводило на мысль о необходимости в первую очередь разработать комплект утилит и инструментов для работы с этими файлами.

Было решено сконцентрироваться на парсинге демо-файлов, а непосредственно анализ сделать позже, возможно, как проект или курсовую.

В процессе работы выяснилось также, что файлы '`.dem`' не содержат некоторых данных, необходимых для хорошего анализа (к примеру, данные о карте и окружении полностью отсутствовали, данные о перемещении игроков ограничивались координатами каждого игрока с привязкой ко времени). Недостающие данные необходимо достроить самостоятельно, а для этого нужно приложение, обеспечивающее уровень абстракции между анализватором и исходными данными. На таком приложении и была сконцентрирована моя работа.

2 Описание конечного результата

В результате была создана библиотека для python 3, предоставляющая набор инструментов для работы с '`.dem`' файлами и обеспечивающая разделение парсинга и анализа.

Система устроена следующим образом: пользователю предоставляется класс, который работает непосредственно с данными записи. Пользователь регистрирует в классе основную функцию, которая получит подготовленные и обработанные данные и произведет анализ.

Оставлены достаточно большие возможности для расширения функционала.

Поддерживается набор необходимых функций, среди которых:

- Обработка данных в несколько процессов (из-за GIL[1] используется именно многопроцессность, а не многопоточность).

  Каждая запись содержит несколько независимых матчей. Функции анализатора можно запустить параллельно на нескольких матчах.

- Доступ к внутренним событиям игры и эмуляция новых.

[1] См. https://wiki.python.org/moin/GlobalInterpreterLock

Пользователю предоставляется возможность добавить необходимые ему события. При происхождении события в игре информация о нем будет передана в анализватор.

Многие важные события уже добавлены в систему.

- Инструменты для визуализации матчей.

  Можно зарегистрировать объект-визуализатор или использовать готовый.

  Визуализатор наглядно показывает на экране действия, происходящие в игре.

  В функции визуализатора также входит обеспечение основной петли ввода-вывода (main loop) для визуализации. Таким образом можно управлять процессом анализа, достаточно лишь добавить в интерфейс визуализатора необходимые элементы управления.

2.1 Подготовка файлов

Класс анализатора принимает на вход не непосредственно файлы '`.dem`', а их подготовленные версии '`.demsd`' (от "demo source data").

Это связано с обеспечением многопоточности: исходный файл разделяется на несколько файлов, чтобы обеспечить одновременный доступ из нескольких процессов.

Непосредственно подготовка файлов производится отдельным конвертором из '`.dem`' в '`.demsd`'.

2.2 Процесс работы с библиотекой

Как отмечалось ранее, весь пользовательский интерфейс сосредоточен в классе анализатора.

Простейший пример работы с данным классом приведен ниже:

```
# coding=utf-8

from analyzer import Analyzer

a = Analyzer('./test.demsd')

# Регистрируем основную функцию для анализатора.
# Для каждого матча будет запущена данная функция.
@a.analyzer
def analyze(processor):
    # processor - iterable, класс, содержащий
    # в себе все необходимые для анализа данные.
    # Экземпляр этого класса создается для каждого
    # обрабатываемого матча.
    # В этом экземпляре содержится вся информация,
```

```
    # доступная пользователю. Гарантируется, что
    # к этому классу нет доступа из других потоков/процессов.

    bullets = 0
    # Цикл по всем кадрам записи
    for global_scope, frame_scope in processor:
        # global_scope - метаданные матча, переменная живет на
            протяжении всего анализа
        # frame_scope - данные кадра, переменная сбрасывается
            каждый кадр
        if 'WeaponFired' in frame_scope.events:
            bullets += len(frame_scope.events['WeaponFired'])
    return bullets

# Вызывается после завершения каждой задачи
# (см. multiprocessing)
@a.callback
def cb(x):
    print(x)

# Вызывается после завершения задачи с ошибкой
@a.error_callback
def err(e):
    raise e

# Запустить анализ всех матчей
# (по умолчанию - 4 процесса)
a.run_analyzer()

# Закрыть пул процессов
a.close()

# Присоединить работающие задачи
a.join()
```

Данный код посчитает количество выстрелов за каждый матч.

2.3 Процессоры данных и события

Для подготовки данных существуют процессоры данных и события.

Перед запуском симуляции запускаются препроцессоры. К примеру, класс визуализации может зарегистрировать препроцессор, открывающий графический интерфейс.

Перед каждым шагом обработки на входных данных запускаются кадровые процессоры (frame_processor). Кадровые процессоры подготавливают данные для

пользователя и генерируют события.

Основной смысл кадровых процессоров в том, что они лишь подготавливают данные, не выполняя анализ.

В данный момент на основе исходных данных генерируется следующая информация:

- frame_scope.tick_data — основная информация по кадру
- frame_scope.tick_data.state — статус каждого игрока:
  - pos x, y, z — позиция игрока
  - view_x, view_y — направление взгляда игрока
  - vel x, y, z — скорость игрока
  - weapon — активное оружие
  - ammo — количество патронов основного оружия
  - weapons — список всего доступного оружия
  - hp — уровень здоровья
  - alive — жив ли игрок
  - stunned — игрок испытывает последствия от светошумовой гранаты
  - sneaks (TODO!) — игрок крадется
  - crouched (TODO!) — игрок пригнулся
  - team — команда (T/CT)
- frame_scope.tick_data.time — времени прошло после начала игры
- frame_scope.tick_data.dt — времени прошло после предыдущего кадра
- global_scope.history — предыдущие кадры

Каждый процессор — это функция, в которую передаются следующие объекты:

- processor — экземпляр класса, запускающего процессоры.[2]
- global_scope — глобальное хранилище. Доступно всем процессорам и событиям, никогда не сбрасывается.
- local_scope — хранилище локальных данных. Оно доступно всегда доступно процессору, при этом для каждого процессора оно свое.
- frame_scope — хранилище информации о кадре. Доступно всем процессорам и событиям, сбрасывается каждый кадр.

[2]Произошла небольшая путаница в названиях. Есть класс процессора, который получает на вход данные из файла демо. Этот класс запускает на полученных данных препроцессоры, кадровые процессоры, обработчики событий и постпроцессоры. В следующих версиях я это все переименую.

Процессоры могут изменять состояние системы, всвязи с чем определен порядок их выполнения: они выполняются точно в том же порядке, в котором были зарегистрированы в системе.

Вот пример некоторых процессоров, доступных в системе:

```
# Расчитывает переменную frame_scope.tick_data.dt
@a.frame_processor
def _frame_processor_dt(processor, global_scope, local_scope,
    frame_scope):
    if 'previous_time' in dir(local_scope):
        frame_scope.tick_data['dt'] = (
            frame_scope.tick_data['time'] -
            local_scope.previous_time)
    else:
        frame_scope.tick_data['dt'] = None
    local_scope.previous_time = frame_scope.tick_data['time']
```

```
from collections import deque

# Хранит историю последних пяти (по умолчанию) кадров
@a.frame_processor
def _frame_processor_history(processor, global_scope,
    local_scope, frame_scope):
    global_scope.history.append(local_scope.previous)
    local_scope.previous = frame_scope
    if ('max_history_length' in dir(processor) and
            len(global_scope.history) > processor.
                max_history_length):
        global_scope.history.popleft()

@_frame_processor_history.setup
def _frame_processor_history_setup(processor, global_scope,
    local_scope):
    global_scope.history = deque()
    local_scope.previous = None
    processor.max_history_length = 5
```

Каждый процессор может генерировать события:

```
@a.frame_processor
def _frame_processor_team(processor, global_scope, local_scope,
    frame_scope):
    if global_scope.history[-1] is not None:
        # Получить все id пользователей на прошлом кадре
```

```
        prev_player_ids = set(global_scope.history[-1].tick_data
            ['state'].keys())
    else:
        prev_player_ids = set()
    # Получить все id пользователей на текущем кадре
    cur_player_ids = set(frame_scope.tick_data['state'].keys())
    for player_id in prev_player_ids - cur_player_ids:
        processor.emit_event('PlayerExited', frame_scope,
            {'player': player_id})
    for player_id in cur_player_ids - prev_player_ids:
        processor.emit_event('PlayerJoined', frame_scope,
            {'player': player_id})
```

События регистрируются в системе и функции-обработчики событий запскаются сразу после того, как последний кадровый процессор завершил работу.

Сами обработчики событий также могут генерировать события, однако их основное назначение — зарегистрировать каждое событие и предоставить о нем информацию пользователю. Регистрация происходит путем добавления данных о событии в словарь '`frame_scope.events`'.

Также частный случай использования событий — система визуализации, отвечающая на каждое событие отображением этого события на экране.

К примеру:

```
@a.event_processor('PlayerKilled')
def _event_processor_player_killed(processor, global_scope,
    local_scope, frame_scope, event_data):
    # Послать команду в процесс отрисовки
    processor.visualizer_remote.set_player_alive(
        event_data['player'], False)
```

Как видно, обработчики событий принимают на вход дополнительный аргумент — данные о событии. В примере с присоединением/выходом игроков это словарь '`{'player': player_id}`'.

На данный момент поддерживаются следующие события:

- WeaponFired — выстрел
- Grenade — бросок любой гранаты
- HEGrenade — бросок разрывной гранаты
- SmokeGrenade — бросок дымовой гранаты
- FlashGrenade — бросок светошумовой гранаты
- PlayerDied — смерть игрока

- PlayerReborn — возрождение игрока (происходит в играх типа deathmatch и при начале нового раунда)
- RoundStarted — новый раунд
- PlayerJoined — новый игрок подключился
- PlayerExited — игрок отключился
- PlayerSwitchTeam — игрок сменил команду
- VisualContact — игрок увидил другого игрока (эта фича работает весьма медленно, см. предложения по улучшениям)
- VisualContactLost — игрок потерял контакт с другим игроком
- BombPlanted — установлена бомба
- BombFound — игрок нашел бомбу (услышал пиканье, но не обязательно точно знает, где она)
- BombDefused — бомба обезврежена
- BombDefuseFailed — обезвреживание бомбы сорвано, игрока отвлекли
- TODO: события, связанные со звуком
- TODO: события, связанные с заложниками

2.4 Визуализатор

Для класса анализатора можно задать визуализатор. Задача визуализатора — нарисовать на экране все данные, необходимые для анализа игры и дебага анализирующего кода.

Технически, визуализатор состоит из двух классов, '`main`' и '`remote`'. '`main`' — синглтон, организующий основную петлю ввода-вывода интерфейса, а '`remote`' — класс, экземпляры которого передаются в процессы обработки, где обрабатывают события и направляют в '`main`' команды для визуализации.

Из коробки доступен визуализатор '`TopViewVisualizer`':

```
# coding=utf-8

from analyzer import Analyzer
from visualization.topview import TopViewVisualizer

a = Analyzer('./data/demos/test.demsd')

a.set_visualizer(TopViewVisualizer)
```

```
# ...

# Запустить анализ всех матчей
a.run_analyzer()

# Запустить петлю ввода-вывода интерфейса
a.run_main_loop()

# После завершения петли интерфейса (все окна закрыты)
# закрыть пул процессов
a.close()
a.join()
```

Необходимо отметить, что визуализатор не умеет подстраиваться под скорость игры. Таким образом, если вы хотите посмотреть игру без ускорения времени, нужно использовать '`time.sleep`' в основной функции анализатора.

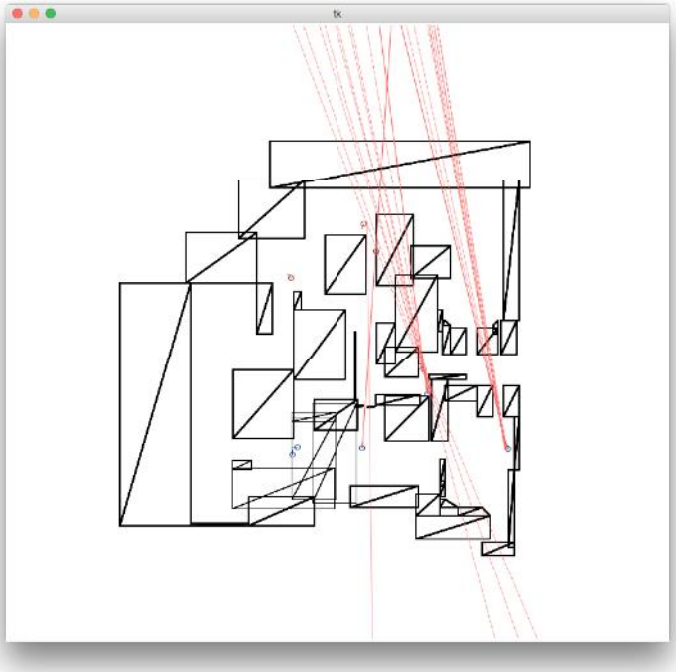

Рис. 1: Визуализатор отображает перестрелку

3 Обзор внешнего API

В данной части дан разбор интерфейса, представленного в классе '`analyzer.Analyzer`'. Подробности реализации тех или иных функций находятся в следующей части.

3.1 Классы ошибок библиотеки

class analyzer.MapError(RuntimeError)

> Нет данных о карте, указанной в демо файле.
> Обратите внимание на '`analyzer.map_search_path`'.

class analyzer.AnalyzerError(RuntimeError)

> Возникает при попытке неправильного использования API. К примеру, попытка изменить количество процессов на уже работающем пуле приведет к '`analyzer.AnalyzerError`'.

class analyzer.ProcessorError(AnalyzerError)

> Ошибка процессора данных (того, что предоставляет итератор на вход основной функции анализатора).
> Возникает при попытке загрузить сломанный '`.demsd`' файл.
> Также при загрузке файла могут возникать ошибки '`IOError`' и '`zipfile.BadZipFile(Exception)`'.

class parsers.error.ParsingError(RuntimeError)

class parsers.vmf.VMFError(ParsingError)

> Ошибка в исходном файле карты. Однако вероятно, что ошибка в парсере файлов '`vmf`'.

3.2 Основной класс

class analyzer.Analyzer

> Основной класс при работе с данной библиотекой.

Analyzer.map_search_path = './data/maps/'

> Путь по умолчанию для поиска файлов карт.
> TODO: превратить этот путь в списсок возможных путей.

Analyzer.analyzer.__init__(self, path)

> **path** — путь к файлу '`.demsd`'. Из-за бага (см. http://stackoverflow.com/questions/5624669/) в библиотеке '`zipfile`' невозможно использовать потоки ввода-вывода в качестве входа.

3.3 Регистрация обработчиков

Analyzer.preprocessor(self, f)

> Декоратор для регистрации препроцессоров.
> **f** — '`callable(processor, global_scope)`', функция-препроцессор, вызывается для каждого матча перед началом обработки кадров и подготавливает данные в '`global_scope`'.

Analyzer.frame_processor(self, f)

> Декоратор для регистрации процессоров кадров.
> **f** — '`callable(processor, global_scope, local_scope, frame_scope)`', функция-процессор, вызывается для каждого кадра.
> После регистрации в '`f`' становятся доступны декораторы '`f.setup`' и '`f.teardown`'.
> **f.setup** — декоратор, регистрирующий функцию, которая будет вызвана перед первым вызовом '`f`': '`callable(processor, global_scope, local_scope)`'.
> **f.teardown** — декоратор, регистрирующий функцию, которая будет вызвана после последнего вызова '`f`': '`callable(processor, global_scope, local_scope)`'.

```
@a.frame_processor
def f(processor, global_scope, local_scope, frame_scope):
    # ...

@f.setup
def f_setup(processor, global_scope, local_scope):
    # ...

@f.teardown
def f_teardown(processor, global_scope, local_scope):
    # ...
```

```
Analyzer.event_processor(self, event)
    return event_decorator(f)
```

> Декоратор для регистрации обработчиков событий.
> **event** — названис события.
> **f** — 'callable(processor, global_scope, local_scope, frame_scope, event_data)', функция-процессор, вызывается всякий раз, как процессор сообщает о событии. После регистрации в 'f' становятся доступны декораторы 'f.setup' и 'f.teardown'.
> **f.setup** — декоратор, регистрирующий функцию, которая будет вызвана перед первым вызовом 'f': 'callable(processor, global_scope, local_scope)'.
> **f.teardown** — декоратор, регистрирующий функцию, которая будет вызвана после последнего вызова 'f': 'callable(processor, global_scope, local_scope)'.

```
@a.event_processor('EventName')
def f(processor, global_scope, local_scope, frame_scope,
    event_data):
    # ...

@f.setup
def f_setup(processor, global_scope, local_scope):
    # ...

@f.teardown
def f_teardown(processor, global_scope, local_scope):
    # ...
```

```
Analyzer.postprocessor(self, f)
```

> Декоратор для регистрации постпроцессоров.
> **f** — 'callable(processor, global_scope)', функция-постпроцессор, вызывается для каждого матча после последнего кадра и уничтожает данные в 'global_scope' (к примеру, закрывает файлы и т. д.).

3.4 Регистрация основных функций

```
Analyzer.analyzer(self, f)
```

> Декоратор для регистрации основной функции анализа.
> **f** — 'callable(processor)'
> Подробнее про 'processor' см. ниже.

`Analyzer.callback(self, f)`

> Декоратор для регистрации коллбека, который запускается после завершения основной функции.
> **f** — '`callable(x)`', где '`x`' — результат работы основной функции.
> Подробнее см. '`multiprocessing`'
> https://docs.python.org/3/library/multiprocessing.html
> **raises 'AnalyzerError'**: невозможно зарегистрировать коллбэк дважды.

`Analyzer.error_callback(self, f)`

> Декоратор для регистрации коллбека, который запускается при возникновении ошибки в основной функции.
> **f** '`callable(e)`', где '`e`' объект ошибки.
> Подробнее см. '`multiprocessing`' —
> https://docs.python.org/3/library/multiprocessing.html
> **raises 'AnalyzerError'**: невозможно зарегистрировать коллбэк ошибки дважды.

`Analyzer.set_visualizer(self, visualizer)`

> Регистратор визуализатора.
> **f** '`callable(e)`', где '`e`' объект ошибки.
> Подробнее про визуализаторы см. «обзор архитектуры»
> **raises 'AnalyzerError'**: невозможно зарегистрировать визуализатор дважды.

3.5 Настройка пула процессов

`pool_size = 4`

> Property, количество процессов в пуле.
> **raises 'AnalyzerError'**: невозможно изменить размер пула, если он уже запущен.

3.6 Запуск анализа

```
Analyzer.run_analyzer(self, mode='all', skip_silently=False)
```

> Собственно, то, что запускает анализ.
> Происходит поиск карты и ее загрузка.
> Запускается пул процессов и визуализатор, после чего пастройка становится невозможной.
> TODO: добавить в функции регистрации процессоров, событий и коллбэков проверку, если пул уже запущен, запретить регистрацию.
> Для каждого матча генерируется свой '`Processor`', который передается как кргумент основной функции.
> **mode** — '`all|single|[list]`', '`all`' запускает все матчи сразу, '`single`' запускает нулевой матч, при передаче списка матчей запускаются матчи из списка.
> **skip_silently** если матч уже запускался, не выдавать ошибку, а просто пропустить его.
> **raises '`AnalyzerError`'**: матч не найдем в списке.
> **raises '`AnalyzerError`'**: матч уже запускался (только если '`skip_silently==False`').

```
Analyzer.running_processes = list()
```

> Read-only property, содеожит список всех обрабатываемых и обработанных матчей. Используется, чтобы отслеживать дублирующиеся запуски анализа па одном и том же матче.

```
Analyzer.run_main_loop(self)
```

> Запускает основную петлю ввода-вывода визуализатора.
> В зависимости от реализации визуализатора, может заблокировать основной процесс, в т. ч. не гарантии, что он будет разблокирован (коллбэки запустятся отдельным процессом).

3.7 Управление процессами после запуска

```
Analyzer.close(self)
```

> Закрывает возможность запускать новые матчи, закрывает пул процессов.

```
Analyzer.join(self)
```

Ожидает, пока все матчи будут обработаны. На это время блокирует основной процесс.
Может быть выполнена только после вызова '`Analyzer.close`'.

```
Analyzer.close(self)
```

Принудительно завершает все процессы анализа. Возможно возникновение ошибок в логике (в т. ч. в визуализаторе), так как память будет освобождена, а запись в очереди межпроцессной связи будет прервана, что может привести к поступлению некорректных данных в основной процесс.
Может быть выполнена только после вызова '`Analyzer.close`'.

3.8 Processor

Экземпляры класса '`Processor`' создаются в методе '`Analyzer.run_analyzer`', однако работать с ними все равно пользователю, так что здесь приводится документация того, что понадобится при работе.

```
class analyzer.Processor
```

Основной класс, предоставляющий данные внутри функции анализа.

```
Processor.global_scope = object()
```

Матаданные матча, переменная живет на протяжении всего анализа.

```
Processor.__iter__(self)
    return self
```

```
Processor.__next__(self)
```

Подготовить информацию о следующем кадре, запустить процессоры и обработчики событий, вернуть информацию.
returns '`self.global_scope`', '`frame_scope`'

`Processor.emit_event(self, name, frame_scope, event_data)`

> Добавить событие на очередь в обработку.
> Событие будет обработано, как только закончится обработка всех процессоров для текущего кадра. Если событие было сгенерировано вне процессора или обработчика события, оно будет отнесено к следующему кадру и будет обработано позже.
> **name** — имя события, hashable.
> **frame_scope** — контекст текущего кадра. Его приходится передавать, так как он не хранится в глобальном состоянии.
> **Осторожно:** если событие было сгенерировано вне процессора или обработчика события, оно будет отнесено к следующему кадру и будет обработано позже, однако контекст кадра не изменится! Это связано с тем, что события не должны генерироваться вне процессоров или обработчиков событий. Это не баг и даже не фича. Хотя, кто знает...
> **frame_scope** — все, что нужно знать обработчику события о событии. Как правило, словарь (‘`dict()`’).

4 Обзор архитектуры

В данном разделе находится более подробное описание взаимодействия основных компонент системы.

С целью не перегружать отчет деталями, оставлены только ключевые компоненты — классы ‘`Analyzer`’, ‘`Processor`’, визуализация.

Не рассмотрены геометрия, парсеры файлов, утилиты (машины состояний, деревья для геометрии и пр.).

4.1 Взаимодействие анализатора, процессора и визуализации

Про анализатор было подробно написано в предыдущей части.

После первого запуска ‘`Analyzer.run_main_loop`’ анализатор запускает пул процессов, используя свои настройки.

Функции обработчики каждого матча добавляются в пул, где и выполняются.

> Необходимо отметить, что для передачи данных между процессами используется ‘`pickle`’, который делает дамп переменных класса/функции/генератора. Для того, чтобы процесс смог принять такие данные, ему нужна возможность импортировать классы, в которых те хранятся.
>
> Это ограничивает использование замыканий и фабрик (фабрики визуализаторов были бы уместны).
>
> Кроме того, это накладывает трудности на передачу данных. Сейчас приходится копировать данные о карте и обрабатываемом матче в соседние процессы, что не совсем эффективно (хотя там не так много данных).

Проблему можно обойти, написав соответствующие классы в thread-safe стиле с использованием структур 'ctypes' (подробнее см. замечания по реализации).

Более подробно, '`Analyzer.run_main_loop`' делает следующее:

1. Если это первый запуск, подготавливаются настройки, запускается пул, блокируется возможность изменить настройки.
2. Если указан визуализатор:
   (a) Если это первый запуск, запускается метод '`setup`' класса '`main`' визуализатора. Этот метод инициализирует визуализатор и регистрирует в системе необходимые обработчики событий.
   (b) Для каждого матча создается экземпляр '`multiprocessing.Pipe`', обеспечивающий сообщение между основной петлей визуализатора и его удаленными частями.
   (c) Для каждого матча запускается метод '`__call__`' класса '`main`' визуализатора, получающий на вход '`multiprocessing.Pipe`'. Этот метод должен вернуть экземпляр класса '`remote`' визуализатора. Этот экземпляр будет передан в процесс, где происходит анализ. Он всегда доступен в '`processor.visualizer_remote`'.
3. Используя только что созданный визуализатор, создается экземпляр класса '`Processor`'. Он получает на вход данные о матче, зарегистрированные процессоры и обработчики, удаленную часть визуализатора.
4. В пул добавляется функция '`analyzer`' с единственным аргументом — ранее созданным экземпляром класса '`Processor`'.
5. Дальнейшая работа происходит, как только пользователь вызывает метод '`__next__`' процессора.

Далее пользователю предлагается использовать полученный объект в качестве '`iterable`'.

При каждом новом запуске '`__next__`' происходит следующее:

1. Если это первый запуск, считываются метаданные матча (секция '`#Head`')
2. Если это первый запуск, запускаются препроцессоры и функции '`setup`', которые на самом деле лежат в полях '`setup_call`' соответствующих процессоров и обработчиков (если вы помните, имя '`setup`' занято декоратором).
3. Считывается очередной кусок данных.
4. Запускаются процессоры кадров.

5. Если в очередь событий после запуска процессоров попало что-то, то, пока очередь не пуста, запускаются обработчики событий.

6. В частности, при запуске обработчиков, зарегистрированных визуализатором, те должны использовать свой экземпляр '`multiprocessing.Pipe`', чтобы передать сообщение о событии в основной процесс. Визуализатор в своей петле ввода/вывода должен регулярно проверять наличие таких сообщений и отображать их на экран.

7. Если возникла ошибка '`StopIteration`', запускаются постпроцессоры и функции '`teardown`', которые на самом деле лежат в полях '`teardown_call`' соответствующих процессоров и обработчиков. После эего продолжается распространение ошибки.

4.2 Порядок запуска пре- и пост- процессоров, методов setup и teardown

Отдельно упомянем порядок вызовов инициализаторов (препроцессоры, '`setup`') и деструкторов (постпроцессоры, '`teardown`').

В начале запускаются препроцессоры в порядке их регистрации.

Потом запускаются методы '`setup`' событий. Так как все события хранятся в словаре, нет гарантий очередности запуска инициализаторов между событиями. Однако для каждого события хранится список их обработчиков в порядке регистрации. В этом порядке и выполняются '`setup`'.

Т. е. если мы зарегистрировали '`f1`' и '`f2`' на обработку события 1, а '`g`' — на обработку события 2, то можно гарантировать, что '`f1`' выполнится перед '`f2`', а вот про их порядок отностиельно '`g`' сказать ничего нельзя.

После запуска инициализаторов событий запускаются инициализаторы процессоров кадров в том порядке, в каком они были зарегистрированы.

Деструкторы запускаются в обратном порядке.

Первым делом запускаются методы '`teardown`' процессоров кадров в обратном порядке к тому, в каком они были зарегистрированы.

Потом идут '`teardown`' событий с той лишь разницей, что теперь '`f2`' выполнится перед '`f1`'.

После запускаются постпроцессоры, опять-таки, в обратном порядке.

5 Обзор формата файлов для анализатора

Всего парсер умеет работать с двумя ворматами: '`.demsd`' и '`.vmf`'.

5.1 .vmf — valve map format

Файлы '`.vmf`' — сохраненные из редактора карты для движка source.

Структура файла достаточно проста, однако есть некоторые подводные камни.

Вот пример простого файла, описывающего куб:

```
world
{
    ...
    solid
    {
        "id" "2"
        side
        {
            "id" "1"
            "plane" "(0 64 0) (64 64 0) (64 0 0)"
            "material" "/TOOLSNODRAW"
            ...
        }
        side
        {
            "id" "2"
            "plane" "(0 0 -512) (64 0 -512) (64 64 -512)"
            "material" "/TOOLSNODRAW"
            ...
        }
        ... [всего 6 плоскостей]
    }
}
```

Основная проблема заключается в том, что объекты (solids) описываются не вершинами и треугольниками, а плоскостями. Условно говоря, в начале у на есть все пространство $\mathbb{R}^3$, и мы отсекаем от него плоскостями куски, оставляя в итоге выпуклый многогранник.

Соответственно, приходится проверять принадлежность каждой точки каждой плоскости.

В целом, работает не плохо, но на маленьких фигурах случаются ошибки.

5.2 .demsd — подготовленные файлы демоверсий (.dem)

Файлы '`.demsd`' получаются из файлов '`.dem`' путем прогона последних через специальную утилиту, идущую в комплекте.

Каждый файл '`.demsd`' представляет из себя zip-архив со следующим содержимым:

- файл '`head`' — json файл, содержащий названис карты, список матчей и другую полезную информацию.
- файлы матчей из списка. Имеют следующий простой формат:

```
#SectionName
{
    "json": "data",
    "time": 3371.915,
    ...
}
#SectionName
...
```

Первая секция всегда '`#Head`'. Последующие секции содержат данные о событиях и кадрах. Все события накапливаются в буфере (грубо говоря, для них вызывается '`Processor.emit_event`', который их и накапливает). Как только встречается '`#Tick`', на его основе создается '`local_scope`' и запускаются процессоры и обработчики.

6 Замечания по реализации и дальнейшее развитие

В тексте отчета присутствует несколько заметок '`TODO`'. В комментариях в коде их еще больше, но они все мелкие.

Отдельно следует отметить некоторые проблемы, появившиеся с добавлением многопроцессности и проблемы с геометрией.

6.1 Лишние копирования при передаче данных между процессами

В двух местах происходят лишние копирования данных для передачи их в другой поток. Это геометрия карты и данные для анализа.

Так как и то и другое в процессе работы программы не изменяется, можно смело давать доступ к одним и тем же данным для всех процессов.

Для обеспечения этого данные для анализа нужно считывать уже в процессе анализа, а не до него. Этого не сложно достичь, изменив парсер '`.demsd`', добавив в него еще один класс, наподобие визуализатора.

С геометрией же все сложнее. Все данные хранятся в дереве квадрантов (quadtree) и поделить его память мажду потоками можно только при использовании низкоуровневых структур из C (модуль '`ctypes`').

Это коррелирует со второй проблемой.

6.2 Низкая скорость работы геометрии

Даже после оптимизации с использованием quadtree, работа модуля геометрии, ответственного за события визуального контакта (а в будующем и за анализ маршрутов), остается крайне медленной.

Это особенность python — тут несколько медленные циклы.