npc-engine

None

None

None

Table of contents

1. Welcome

NPC engine is an inference engine that allows you to design game NPC AI using natural language.

It uses state of the art deep learning models and allows you to:

- Generate NPC lines based on the natural language descriptions of a character (e.g. his persona, location).
- Voice those lines with multiple voices (current model has 127)
- Trigger discrete actions based on semantic similarity between an action description and a player or NPC line.
- Create an API to your own deep learning models.

1.1 Use npc-engine

Unity | **Unreal Engine 4** | **Godot** | **Other**

1.2 Extend npc-engine

Export model using existing API | **Add new API**

2. Inference Engine

2.1 Overview

This is the documentation for inference 0MQ server. It describes how it works internally, how to integrate with it and how to extend it.

2.1.1 How does it work

Inference server is build around 0MQ REQ/REP sockets and is using JSON-RPC 2.0 protocol to communicate.

When starting a server models path must be provided

```
cli.exe run --models-path models
```

When the server starts it scans the folder for any valid models, loads them and exposes their API.

Each model's API is defined in API classes.
Server exposes methods that are listed in `API_METHODS` class variable.

You can find a description of default models available in Default Models section.

The specifics of how model is loaded and how inference is done is defined in specific model classes

> Warning

Right now only single model of the same API type can be loaded.

When the server is started you can run model apis via JSON-RPC requests.

`{ "method": "do_smth", "params": ["hello"], "jsonrpc": "2.0", "id": 0 }`
will result in call to `some_model.do_smth('hello')` on the server.

2.1.2 Creating an integration

A checklist for a new integration would be to:

- Create a class that manages npc-engine subprocess (starts, terminates, checks if it's alive).
- Create a connection class that talks to npc-engine.
- Review API classes and wrap JSON-RPC requests into the native functions.

2.2 Running The Server

First lets get the `npc-engine`.

You can get it from

- Releases page
- Resulting folder after following build instructions

Inside the *npc-engine* folder *cli.exe* can be found. This is the CLI interface to `npc-engine` server.

You can check all the possible commands via:

```
cli.exe --help
```

To start the server create models directory:

```
mkdir models
```

and execute cli.exe with run command

```
cli.exe run --models-path models --port 5555
```

This will start a server but if no models were added to the folder it won't expose any API.

You can download default models via

```
cli.exe download-default-models --models-path models
```

See descriptions of the default models in Default Models section.

> NOTE

Model API examples can be found in `npc-engine\tests\integration`.
If you don't need any specific model functionality just don't add this model to your *models* folder.

Now lets test npc-engine with this example request from python:

```
import zmq
context = zmq.Context()

#  Socket to talk to server
print("Connecting to npc-engine server")
socket = context.socket(zmq.REQ)
socket.RCVTIMEO = 2000
socket.connect("tcp://localhost:5555")
request = {
    "jsonrpc": "2.0",
    "method": "compare",
    "id": 0,
    "params": ["I will help you", ["I shall provide you my assistance"]],
}
socket.send_json(request)
message = socket.recv_json()
print(f"Response message {message}")
```

2.3 Build From Source

2.3.1 Build on Windows

CREATE VIRTUALENV AND ACTIVATE IT

```
python3 -m venv npc-engine-venv
.\npc-engine-venv\activate.bat
```

INSTALL DEPENDENCIES

```
pip install -e .[dev,dml]
```

(OPTIONAL) COMPILE, BUILD AND INSTALL YOUR CUSTOM ONNX PYTHON RUNTIME"

Build instructions can be found here

(OPTIONAL) RUN TESTS

- Download models to run tests against into `npc-engine\resources\models`.
  You can use default models from here
- Run tests with `tox`

COMPILE TO EXE WITH

```
pyinstaller --additional-hooks-dir hooks --exclude-module matplotlib --exclude-module jupyter --exclude-module torch --exclude-module torchvision .\npc-engine\cli.
```

2.4 API Classes

API class is an abstract class that corresponds to a certain task a model should perform (e.g. text-to-speech or chatbot) and defines interface methods for such a task as well as abstract methods for specific models to implement.

All API classes are children of the Model class that handles registering model implementations and loading them.

> Important

It also should list the methods that are to be exposed as API via API_METHODS class variable.

> Important

To be discovered correctly api classes must be imported into npc_engine.models module

2.4.1 Existing APIs

These are the existing API classes and corresponding API_METHODS:

`npc_engine.models.chatbot.chatbot_base.ChatbotAPI (Model)`

Abstract base class for Chatbot models.

`generate_reply(self, context, *args, **kwargs)`

Format the model prompt and generate response.

Parameters:

Name	Type	Description	Default
`context`	`Dict[str, Any]`	Prompt context.	*required*

Returns:

Type	Description
`str`	Text response to a prompt.

`get_context_fields(self)`

Return context template used for formatting model prompt.

Returns:

Type	Description
`List[str]`	A template context dict with empty fields.

`get_prompt_template(self)`

Return prompt template string used to render model prompt.

Returns:

Type	Description
`str`	A template string.

`npc_engine.models.similarity.similarity_base.SimilarityAPI (Model)`

Abstract base class for text similarity models.

`cache(self, context)`

Compare a query to the context.

Parameters:

Name	Type	Description	Default
`query`		A sentence to compare.	*required*
`context`	`List[str]`	A list of sentences to compare to. This will be cached if caching is enabled	*required*

Returns:

Type	Description
	List of similarities

`compare(self, query, context)`

Compare a query to the context.

Parameters:

Name	Type	Description	Default
`query`	`str`	A sentence to compare.	*required*
`context`	`List[str]`	A list of sentences to compare to. This will be cached if caching is enabled	*required*

Returns:

Type	Description
`List[float]`	List of similarities

npc_engine.models.tts.tts_base.TextToSpeechAPI (Model)

Abstract base class for text-to-speech models.

get_speaker_ids(self)

Get list of available speaker ids.

Returns:

Type	Description
`List[str]`	The return value. True for success, False otherwise.

tts_get_results(self)

Retrieve the next chunk of generated speech.

Returns:

Type	Description
`Iterable[numpy.ndarray]`	Next chunk of speech in the form of f32 ndarray.

tts_start(self, speaker_id, text, n_chunks)

Initiate iterative generation of speech.

Parameters:

Name	Type	Description	Default
`speaker_id`	`str`	Id of the speaker.	*required*
`text`	`str`	Text to generate speech from.	*required*
`n_chunks`	`int`	Number of chunks to split generation into.	*required*

2.4.2 Creating new APIs

You can use this dummy API example to create your own:

```
from npc_engine.models.base_model import Model

class EchoAPI(Model):
    API_METHODS: List[str] = ["echo"]
    def __init__(self, *args, **kwargs):
        pass

    def echo(self, text):
        return text
```

Dont forget

Import new API to npc-engine.models so that it is discovered. Models that are implemented for the API should appear there too.

2.5 Models

Model classes are specific implementations of API classes. They define the loading process in `__init__` function and all the abstract methods required by the API class to function.

2.5.1 How models are loaded?

ModelManager Scans the models folder. For each discovered subfolder Model.load is called, which tries to read `config.yml` field `model_type`. This field must contain correct model class that was discovered and registered by `Model` parent class, and if it does, this class is instantiated with whole `config.yml` parsed dictionary as parameters.

2.5.2 How is their API exposed?

ModelManager builds a mapping from model's API_METHODS class variable to anonymous functions that call these methods on the model. This dictionary is then served through `json-rpc` protocol implementation.

2.5.3 Existing model classes

`npc_engine.models.chatbot.bart.BartChatbot (ChatbotAPI)`

BART based chatbot implementation class.

This model class requires two ONNX models `encoder_bart.onnx` and `decoder_bart.onnx` that correspond to encoder and decoder from transformers EncoderDecoderModel and a tokenizer.json with huggingface tokenizers definition.

encoder_bart.onnx spec:

```
- inputs:
    `input_ids`
- outputs:
    `encoder_hidden_state`
```

decoder_bart.onnx spec:

```
- inputs:
    `encoder_hidden_state`
    `decoder_input_ids`
- outputs:
    `logits`
```

`__init__(self, model_path, max_steps=100, min_length=2, repetition_penalty=1, bos_token_id=0, eos_token_id=2, pad_token_id=1, sep_token_id=None, *args, **kwargs)` special

Create the chatbot from config args and kwargs.

Parameters:

Name	Type	Description	Default
`model_path`		path to scan for model files (weights and configs)	*required*
`max_steps`		stop generation at this number of tokens	`100`
`min_length`		model can't stop generating text before it's atleast this long in tokens	`2`
`repetition_penalty`		probability coef for same tokens to appear multiple times	`1`
`bos_token_id`		beginning of sequence token id	`0`
`eos_token_id`		end of sequence token id	`2`
`pad_token_id`		padding token id	`1`
`sep_token_id`		token id for separating sequence into multiple parts	`None`

`get_special_tokens(self)`

Retrun dict of special tokens to be renderable from template.

`run(self, prompt, temperature=1.0, topk=None)`

Run text generation from given prompt and parameters.

Parameters:

Name	Type	Description	Default
`prompt`	`str`	Fromatted prompt.	*required*
`temperature`	`float`	Temperature parameter for sampling. Controls how random model output is: more temperature - more randomness	`1.0`
`topk`	`int`	If not none selects top n of predictions to sample from during generation.	`None`

Returns:

Type	Description
	Generated text

`npc_engine.models.similarity.similarity_transformers.TransformerSemanticSimilarity (SimilarityAPI)`

Huggingface transformers semantic similarity.

Uses ONNX export of Huggingface transformers (https://huggingface.co/models) with biencoder architecture. Also requires a tokenizer.json with huggingface tokenizers definition.

model.onnx spec:

```
- inputs:
    `input_ids` of shape `(batch_size, sequence)`
    `attention_mask` of shape `(batch_size, sequence)`
    (Optional) `input_type_ids` of shape `(batch_size, sequence)`
- outputs:
    `token_embeddings` of shape `(batch_size, sequence, hidden_size)`
```

`__init__(self, model_path, metric='dot', pad_token_id=0, *args, **kwargs)` `special`

Create and load biencoder model for semantic similarity.

Parameters:

Name	Type	Description	Default
`model_path`	`str`	A path where model config and weights are	*required*
`metric`	`str`	distance to compute semantic similarity	`'dot'`

`compute_embedding(self, line)`

Compute line embeddings in batch.

Parameters:

Name	Type	Description	Default
`lines`		List of sentences to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding batch of shape (batch_size, embedding_size)

`compute_embedding_batch(self, lines)`

Compute sentence embedding.

Parameters:

Name	Type	Description	Default
`line`		Sentence to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding of shape (1, embedding_size)

`metric(self, embedding_a, embedding_b)`

Similarity between two embeddings.

Embeddings are of broadcastable shapes. (1 or batch_size)

Parameters:

Name	Type	Description	Default
`embedding_a`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*
`embedding_b`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*

Returns:

Type	Description
`ndarray`	Vector of distances (batch_size or 1,)

`npc_engine.models.tts.flowtron.FlowtronTTS (TextToSpeechAPI)`

Implements Flowtron architecture inference.

Paper:
arXiv:2005.05957
Code:
https://github.com/NVIDIA/flowtron

Onnx export script can be found in this fork https://github.com/npc-engine/flowtron.

This model class requires four ONNX models `encoder.onnx`, `backward_flow.onnx`, `forward_flow.onnx` and `vocoder.onnx` where first three are layers from Flowtron architecture (`flow` corresponding to one direction pass of affine coupling layers) and `vocoder.onnx` is neural vocoder.

For detailed specs refer to https://github.com/npc-engine/flowtron.

`__init__(self, model_path, max_frames=400, gate_threshold=0.5, sigma=0.8, smoothing_window=3, smoothing_weight=0.5, *args, **kwargs)` special

Create and load Flowtron and vocoder models.

`get_speaker_ids(self)`

Return available ids of different speakers.

`run(self, speaker_id, text, n_chunks)`

Create a generator for iterative generation of speech.

Parameters:

Name	Type	Description	Default
`speaker_id`	`str`	Id of the speaker.	*required*
`text`	`str`	Text to generate speech from.	*required*
`n_chunks`	`int`	Number of chunks to split generation into.	*required*

Returns:

Type	Description
`Iterator[numpy.ndarray]`	Generator that yields next chunk of speech in the form of f32 ndarray.

2.5.4 Default Models

- **Fantasy Chatbot**

  BartChatbot trained on LIGHT Dataset. Model consumes both self, other personas and location dialogue is happening in.

- **Semantic Similarity sentence-transformers/all-MiniLM-L6-v2**

  Onnx export of sentence-transformers/all-MiniLM-L6-v2.

- **FlowtronTTS with Waveglow vocoder**

  Nvidia's FlowtronTTS architecture using Waveglow vocoder. Weights were published by the authors, this model uses Flowtron LibriTTS2K version.

- **Speech to text NeMo models**

  This model is still heavy WIP it is best to use your platform's of choice existing solutions e.g. UnityEngine.Windows.Speech.DictationRecognizer in Unity.

  This implementation uses several models exported from NeMo toolkit:

  - QuartzNet15x5 for transcription.
  - Punctuation BERT for applying punctuation.
  - Custom transformer for recognizing end of response to the context initialized from all-MiniLM-L6-v2

2.5.5 Creating new models

You can use this dummy model example to create your own:

```
from npc_engine.models.base_model import Model

class EchoModel(ChatbotAPI):

    def __init__(self, model_path:str, *args, **kwargs):
        print("model is in {model_path}")

    def get_special_tokens(self):
        return {}

    def run(self, prompt, temperature=1, topk=None):
        return prompt
```

Dont forget

Import new model to npc-engine.models so that it is discovered.

2.6 Reference

Server for providing onnx runtime predictions for text generation and speech synthesis.

Uses 0MQ REP/REQ sockets with JSONRPC 2.0 protocol.

2.6.1 `cli`

This is the entry point for the command-line interface that starts npc-engine server.

2.6.2 `models` `special`

Module that contains everything related to deep learning models.

For your model API to be discovered it must be imported here

`base_model`

Module with Model base class.

`Model (ABC)`

Abstract base class for managed models.

`__init_subclass__(**kwargs)` `classmethod` `special`

Init subclass where model classes get registered to be loadable.

Source code in `npc_engine/models/base_model.py`

```
def __init_subclass__(cls, **kwargs):
    """Init subclass where model classes get registered to be loadable."""
    super().__init_subclass__(**kwargs)
    cls.models[cls.__name__] = cls
```

`load(path)` `classmethod`

Load the model from the path.

Source code in `npc_engine/models/base_model.py`

```
@classmethod
def load(cls, path: str):
    """Load the model from the path."""
    config_path = os.path.join(path, "config.yml")
    with open(config_path) as f:
        config_dict = yaml.load(f, Loader=yaml.Loader)
    config_dict["model_path"] = path
    model_cls = cls.models[config_dict["model_type"]]
    return model_cls(**config_dict)
```

chatbot special

Chatbot model implementations.

This module implements specific models and wraps them under the common interface for loading and inference.

Examples:

```
from npc_engine.models.chatbot import ChatbotAPI
model = ChatbotAPI.load("path/to/model_dir")
model.generate_reply(context, temperature=0.8, topk=None,)
```

bart

BART based chatbot implementation.

BartChatbot (ChatbotAPI)

BART based chatbot implementation class.

This model class requires two ONNX models `encoder_bart.onnx` and `decoder_bart.onnx` that correspond to encoder and decoder from transformers EncoderDecoderModel and a tokenizer.json with huggingface tokenizers definition.

encoder_bart.onnx spec:

```
- inputs:
    `input_ids`
- outputs:
    `encoder_hidden_state`
```

decoder_bart.onnx spec:

```
- inputs:
    `encoder_hidden_state`
    `decoder_input_ids`
- outputs:
    `logits`
```

`__init__(self, model_path, max_steps=100, min_length=2, repetition_penalty=1, bos_token_id=0, eos_token_id=2, pad_token_id=1, sep_token_id=None, *args, **kwargs)` special

Create the chatbot from config args and kwargs.

Parameters:

Name	Type	Description	Default
`model_path`		path to scan for model files (weights and configs)	*required*
`max_steps`		stop generation at this number of tokens	`100`
`min_length`		model can't stop generating text before it's atleast this long in tokens	`2`
`repetition_penalty`		probability coef for same tokens to appear multiple times	`1`
`bos_token_id`		beginning of sequence token id	`0`
`eos_token_id`		end of sequence token id	`2`
`pad_token_id`		padding token id	`1`
`sep_token_id`		token id for separating sequence into multiple parts	`None`

Source code in `npc_engine/models/chatbot/bart.py`

```
def __init__(
    self,
    model_path,
    max_steps=100,
    min_length=2,
    repetition_penalty=1,
    bos_token_id=0,
    eos_token_id=2,
    pad_token_id=1,
    sep_token_id=None,
    *args,
    **kwargs,
):
    """Create the chatbot from config args and kwargs.

    Args:
        model_path: path to scan for model files (weights and configs)
        max_steps: stop generation at this number of tokens
        min_length: model can't stop generating text before it's atleast
            this long in tokens
        repetition_penalty: probability coef for same tokens to appear multiple times
        bos_token_id: beginning of sequence token id
        eos_token_id: end of sequence token id
        pad_token_id: padding token id
        sep_token_id: token id for separating sequence into multiple parts

    """
    super().__init__(*args, **kwargs)
    self.bos_token_id = bos_token_id
    self.eos_token_id = eos_token_id
    self.sep_token_id = eos_token_id if sep_token_id is None else sep_token_id
    self.pad_token_id = pad_token_id

    sess_options = rt.SessionOptions()
    sess_options.graph_optimization_level = rt.GraphOptimizationLevel.ORT_ENABLE_ALL
    self.encoder_model = rt.InferenceSession(
        os.path.join(model_path, "encoder_bart.onnx"),
        providers=[rt.get_available_providers()[0]],
        sess_options=sess_options,
    )
    self.decoder_model = rt.InferenceSession(
        os.path.join(model_path, "decoder_bart.onnx"),
        providers=[rt.get_available_providers()[0]],
        sess_options=sess_options,
    )
    self.tokenizer = Tokenizer.from_file(os.path.join(model_path, "tokenizer.json"))
    added_tokens_path = os.path.join(model_path, "added_tokens.txt")
    if os.path.exists(added_tokens_path):
        with open(added_tokens_path) as f:
            added_tokens = json.load(f)
        added_tokens = [
            key for key, _ in sorted(list(added_tokens.items()), key=lambda x: x[1])
        ]
    self.tokenizer.add_tokens(added_tokens)
    self.special_tokens = {
        "bos_token": self.tokenizer.decode(
            [bos_token_id], skip_special_tokens=False
        ),
        "eos_token": self.tokenizer.decode(
            [eos_token_id], skip_special_tokens=False
        ),
        "sep_token": self.tokenizer.decode(
            [self.sep_token_id], skip_special_tokens=False
        ),
        "pad_token": self.tokenizer.decode(
            [pad_token_id], skip_special_tokens=False
        ),
        **{
            f"added_token{self.tokenizer.token_to_id(token)}": token
            for token in added_tokens
        },
    }

    self.max_steps = max_steps
    self.min_length = min_length
    self.repetition_penalty = repetition_penalty
```

`get_special_tokens(self)`

Retrun dict of special tokens to be renderable from template.

Source code in `npc_engine/models/chatbot/bart.py`

```
def get_special_tokens(self) -> Dict[str, str]:
    """Retrun dict of special tokens to be renderable from template."""
    return self.special_tokens
```

`run(self, prompt, temperature=1.0, topk=None)`

Run text generation from given prompt and parameters.

Parameters:

Name	Type	Description	Default
`prompt`	`str`	Fromatted prompt.	*required*
`temperature`	`float`	Temperature parameter for sampling. Controls how random model output is: more temperature - more randomness	`1.0`
`topk`	`int`	If not none selects top n of predictions to sample from during generation.	`None`

Returns:

Type	Description
	Generated text

Source code in `npc_engine/models/chatbot/bart.py`

```
    def run(self, prompt: str, temperature: float = 1.0, topk: int = None):
        """Run text generation from given prompt and parameters.

        Args:
            prompt: Fromatted prompt.
            temperature: Temperature parameter for sampling.
                Controls how random model output is: more temperature - more randomness
            topk: If not none selects top n of predictions to sample from during generation.

        Returns:
            Generated text
        """
        print(
            f"""Prompt:
{prompt}
Prompt end
        """
        )
        tokens = self.tokenizer.encode(prompt)
        total = np.asarray(tokens.ids, dtype=np.int64).reshape([1, -1])
        total_enc = self.encoder_model.run(None, {"input_ids": total})[0]

        utterance = np.asarray([self.eos_token_id], dtype=np.int64).reshape([1, 1])

        for i in range(self.max_steps):
            o = self.decoder_model.run(
                None,
                {"encoder_hidden_state": total_enc, "decoder_input_ids": utterance},
            )
            logits = o[0][0, -1, :]

            if i < self.min_length:
                logits[self.eos_token_id] = float("-inf")
            if topk is not None:
                ind = np.argpartition(logits, -topk)[-topk:]
                new_logits = np.zeros(logits.shape)
                new_logits[ind] = logits[ind]
                logits = new_logits
```

```
            probs = scp.softmax(logits / temperature, axis=0)

            token = np.random.choice(np.arange(probs.shape[0]), p=probs)
            token = token.reshape([1, 1])
            utterance = np.concatenate([utterance, token], axis=1)
            if token[0, 0] == self.eos_token_id:
                break
        return self.tokenizer.decode(utterance[0, :].tolist(), skip_special_tokens=True)
```

chatbot_base

Module that implements chatbot model API.

ChatbotAPI (Model)

Abstract base class for Chatbot models.

`__init__(self, template_string, default_context, *args, **kwargs)` special

Initialize prompt formatting variables.

Parameters:

Name	Type	Description	Default
`template_string`	`str`	Template string to be rendered.	*required*
`default_context`	`str`	Context example with empty fields.	*required*

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
def __init__(self, template_string: str, default_context: str, *args, **kwargs):
    """Initialize prompt formatting variables.

    Args:
        template_string: Template string to be rendered.
        default_context: Context example with empty fields.
    """
    self.template_string = template_string
    self.default_context = json.loads(default_context)
    self.template = Template(template_string)
    self.initialized = True
```

`generate_reply(self, context, *args, **kwargs)`

Format the model prompt and generate response.

Parameters:

Name	Type	Description	Default
`context`	`Dict[str, Any]`	Prompt context.	*required*

Returns:

Type	Description
`str`	Text response to a prompt.

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
def generate_reply(self, context: Dict[str, Any], *args, **kwargs) -> str:
    """Format the model prompt and generate response.

    Args:
        context: Prompt context.
        *args
        **kwargs

    Returns:
        Text response to a prompt.
    """
    if not self.initialized:
        raise AssertionError(
            "Can not generate replies before base Chatbot class was initialized"
        )
    prompt = self.template.render(**context, **self.get_special_tokens())
    return self.run(prompt, *args, **kwargs)
```

`get_context_fields(self)`

Return context template used for formatting model prompt.

Returns:

Type	Description
`List[str]`	A template context dict with empty fields.

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
def get_context_fields(self) -> List[str]:
    """Return context template used for formatting model prompt.

    Returns:
        A template context dict with empty fields.
    """
    return self.default_context
```

`get_prompt_template(self)`

Return prompt template string used to render model prompt.

Returns:

Type	Description
`str`	A template string.

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
def get_prompt_template(self) -> str:
    """Return prompt template string used to render model prompt.

    Returns:
        A template string.
    """
    return self.template_string
```

`get_special_tokens(self)`

Return dictionary mapping for special tokens.

To be implemented by child class. Can then be used in template string as fields

Returns:

Type	Description
`Dict[str, str]`	Dictionary of special tokens

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
@abstractmethod
def get_special_tokens(self) -> Dict[str, str]:
    """Return dictionary mapping for special tokens.

    To be implemented by child class.
    Can then be used in template string as fields
    Returns:
        Dictionary of special tokens
    """
    return None
```

`run(self, prompt, temperature=1, topk=None)`

Abstract method for concrete implementation of generation.

Parameters:

Name	Type	Description	Default
`prompt`	`str`	Fromatted prompt.	*required*
`temperature`	`float`	Temperature parameter for sampling. Controls how random model output is: more temperature - more randomness	`1`
`topk`	`int`	If not none selects top n of predictions to sample from during generation.	`None`

Returns:

Type	Description
`str`	Generated text

Source code in `npc_engine/models/chatbot/chatbot_base.py`

```
@abstractmethod
def run(self, prompt: str, temperature: float = 1, topk: int = None) -> str:
    """Abstract method for concrete implementation of generation.

    Args:
        prompt: Fromatted prompt.
        temperature: Temperature parameter for sampling.
            Controls how random model output is: more temperature - more randomness
        topk: If not none selects top n of predictions to sample from during generation.

    Returns:
        Generated text
    """
    return None
```

model_manager

Module that implements management and loading of the models.

ModelManager

Loads the models and creates global API dictionary.

__init__(self, path) special

Create model manager and load models from the given path.

Source code in npc_engine/models/model_manager.py

```
def __init__(self, path):
    """Create model manager and load models from the given path."""
    subdirs = [
        f.path
        for f in os.scandir(path)
        if f.is_dir() and os.path.exists(os.path.join(f, "config.yml"))
    ]
    self.models = [models.Model.load(subdir) for subdir in subdirs]
```

build_api_dict(self)

Build api dict.

Returns:

Type	Description
dict(str,str)	Mapping "method_name" -> callable that will be exposed to API

Source code in npc_engine/models/model_manager.py

```
def build_api_dict(self):
    """Build api dict.

    Returns:
        dict(str,str): Mapping "method_name" -> callable
            that will be exposed to API
    """
    api_dict = {}
    for model in self.models:
        for method in type(model).API_METHODS:
            logger.info(
                f"Registering method {method} for model {type(model).__name__}"
            )
            api_dict[method] = self._build_api_call(model, method)
    return api_dict
```

similarity special

Similarity model implementations.

This module implements specific models and wraps them under the common interface for loading and inference.

Examples:

```
from npc_engine.models.similarity import SimilarityAPI
model = SimilarityAPI.load("path/to/model_dir")
model.compare("hello", ["Hello, world!"])
```

`similarity_base`

Module that implements semantic similarity model API.

`SimilarityAPI (Model)`

Abstract base class for text similarity models.

`__init__(self, cache_size=0, *args, **kwargs) special`

Empty initialization method for API to be similar to other model base classes.

Source code in `npc_engine/models/similarity/similarity_base.py`

```
def __init__(self, cache_size=0, *args, **kwargs) -> None:
    """Empty initialization method for API to be similar to other model base classes."""
    super().__init__()
    self.initialized = True
    self.lru_cache = NumpyLRUCache(cache_size)
```

`cache(self, context)`

Compare a query to the context.

Parameters:

Name	Type	Description	Default
`query`		A sentence to compare.	*required*
`context`	`List[str]`	A list of sentences to compare to. This will be cached if caching is enabled	*required*

Returns:

Type	Description
	List of similarities

Source code in `npc_engine/models/similarity/similarity_base.py`

```
def cache(self, context: List[str]):
    """Compare a query to the context.

    Args:
        query: A sentence to compare.
        context: A list of sentences to compare to. This will be cached if caching is enabled

    Returns:
        List of similarities
    """
    self.lru_cache.cache_compute(
        context, lambda values: self.compute_embedding_batch(values)
    )
```

`compare(self, query, context)`

Compare a query to the context.

Parameters:

Name	Type	Description	Default
`query`	`str`	A sentence to compare.	*required*
`context`	`List[str]`	A list of sentences to compare to. This will be cached if caching is enabled	*required*

Returns:

Type	Description
`List[float]`	List of similarities

Source code in `npc_engine/models/similarity/similarity_base.py`

```
def compare(self, query: str, context: List[str]) -> List[float]:
    """Compare a query to the context.

    Args:
        query: A sentence to compare.
        context: A list of sentences to compare to. This will be cached if caching is enabled

    Returns:
        List of similarities
    """
    embedding_a = self.compute_embedding(query)
    embedding_b = self.lru_cache.cache_compute(
        context, lambda values: self.compute_embedding_batch(values)
    )
    similarities = self.metric(embedding_a, embedding_b)
    return similarities.tolist()
```

`compute_embedding(self, line)`

Compute sentence embedding.

Parameters:

Name	Type	Description	Default
`line`	`str`	Sentence to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding of shape (1, embedding_size)

Source code in `npc_engine/models/similarity/similarity_base.py`

```
@abstractmethod
def compute_embedding(self, line: str) -> np.ndarray:
    """Compute sentence embedding.

    Args:
        line: Sentence to embed

    Returns:
```

```
        Embedding of shape (1, embedding_size)
    """
    return None
```

`compute_embedding_batch(self, lines)`

Compute line embeddings in batch.

Parameters:

Name	Type	Description	Default
`lines`	`List[str]`	List of sentences to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding batch of shape (batch_size, embedding_size)

Source code in `npc_engine/models/similarity/similarity_base.py`

```
@abstractmethod
def compute_embedding_batch(self, lines: List[str]) -> np.ndarray:
    """Compute line embeddings in batch.

    Args:
        lines: List of sentences to embed

    Returns:
        Embedding batch of shape (batch_size, embedding_size)
    """
    return None
```

`metric(self, embedding_a, embedding_b)`

Compute distance between two embeddings.

Embeddings are of broadcastable shapes. (1 or batch_size)

Parameters:

Name	Type	Description	Default
`embedding_a`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*
`embedding_b`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*

Returns:

Type	Description
`ndarray`	Vector of distances (batch_size or 1,)

Source code in `npc_engine/models/similarity/similarity_base.py`

```
@abstractmethod
def metric(self, embedding_a: np.ndarray, embedding_b: np.ndarray) -> np.ndarray:
    """Compute distance between two embeddings.

    Embeddings are of broadcastable shapes. (1 or batch_size)
    Args:
        embedding_a: Embedding of shape (1 or batch_size, embedding_size)
        embedding_b: Embedding of shape (1 or batch_size, embedding_size)

    Returns:
        Vector of distances (batch_size or 1,)
    """
    return None
```

`similarity_transformers`

Module that implements Huggingface transformers semantic similarity.

`TransformerSemanticSimilarity (SimilarityAPI)`

Huggingface transformers semantic similarity.

Uses ONNX export of Huggingface transformers (https://huggingface.co/models) with biencoder architecture. Also requires a tokenizer.json with huggingface tokenizers definition.

model.onnx spec:

```
- inputs:
    `input_ids` of shape `(batch_size, sequence)`
    `attention_mask` of shape `(batch_size, sequence)`
    (Optional) `input_type_ids` of shape `(batch_size, sequence)`
- outputs:
    `token_embeddings` of shape `(batch_size, sequence, hidden_size)`
```

`__init__(self, model_path, metric='dot', pad_token_id=0, *args, **kwargs)` special

Create and load biencoder model for semantic similarity.

Parameters:

Name	Type	Description	Default
`model_path`	`str`	A path where model config and weights are	*required*
`metric`	`str`	distance to compute semantic similarity	`'dot'`

Source code in `npc_engine/models/similarity/similarity_transformers.py`

```
def __init__(
    self,
    model_path: str,
    metric: str = "dot",
    pad_token_id: int = 0,
    *args,
    **kwargs
):
    """Create and load biencoder model for semantic similarity.

    Args:
        model_path: A path where model config and weights are
        metric: distance to compute semantic similarity
    """
    super().__init__(*args, **kwargs)
    sess_options = rt.SessionOptions()
```

```
    sess_options.graph_optimization_level = opt_level.ORT_ENABLE_ALL
    self.model = rt.InferenceSession(
        os.path.join(model_path, "model.onnx"),
        providers=[rt.get_available_providers()[0]],
        sess_options=sess_options,
    )
    input_names = [inp.name for inp in self.model.get_inputs()]
    self.token_type_support = "token_type_ids" in input_names
    self.pad_token_id = pad_token_id
    self.tokenizer = Tokenizer.from_file(os.path.join(model_path, "tokenizer.json"))
    self.tokenizer.enable_padding(
        direction="right",
        pad_id=self.pad_token_id,
        pad_type_id=0,
        pad_token=self.tokenizer.decode(
            [self.pad_token_id], skip_special_tokens=False
        ),
        length=None,
        pad_to_multiple_of=None,
    )
    self.tests = {}
    self.metric_type = metric
```

`compute_embedding(self, line)`

Compute line embeddings in batch.

Parameters:

Name	Type	Description	Default
`lines`		List of sentences to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding batch of shape (batch_size, embedding_size)

Source code in `npc_engine/models/similarity/similarity_transformers.py`

```
def compute_embedding(self, line: str) -> np.ndarray:
    """Compute line embeddings in batch.

    Args:
        lines: List of sentences to embed

    Returns:
        Embedding batch of shape (batch_size, embedding_size)
    """
    ids = (
        np.asarray(self.tokenizer.encode(line).ids)
        .reshape([1, -1])
        .astype(np.int64)
    )
    attention_mask = np.ones_like(ids)
    if not self.token_type_support:
        input_dict = {"input_ids": ids, "attention_mask": attention_mask}
    else:
        input_dict = {
            "input_ids": ids,
            "attention_mask": attention_mask,
            "token_type_ids": np.zeros_like(ids),
        }
    outp = self.model.run(None, input_dict)
    return self._mean_pooling(outp, attention_mask)
```

`compute_embedding_batch(self, lines)`

Compute sentence embedding.

Parameters:

Name	Type	Description	Default
`line`		Sentence to embed	*required*

Returns:

Type	Description
`ndarray`	Embedding of shape (1, embedding_size)

Source code in `npc_engine/models/similarity/similarity_transformers.py`

```
def compute_embedding_batch(self, lines: List[str]) -> np.ndarray:
    """Compute sentence embedding.

    Args:
        line: Sentence to embed

    Returns:
        Embedding of shape (1, embedding_size)
    """
    tokenized = self.tokenizer.encode_batch(lines)
    ids = np.stack([np.asarray(encoding.ids) for encoding in tokenized]).astype(
        np.int64
    )
    attention_mask = np.stack(
        [np.asarray(encoding.attention_mask) for encoding in tokenized]
    ).astype(np.int64)
    if not self.token_type_support:
        input_dict = {"input_ids": ids, "attention_mask": attention_mask}
    else:
        input_dict = {
            "input_ids": ids,
            "attention_mask": attention_mask,
            "token_type_ids": np.stack(
                [np.asarray(encoding.type_ids) for encoding in tokenized]
            ).astype(np.int64),
        }
    outp = self.model.run(None, input_dict)
    return self._mean_pooling(outp, attention_mask)
```

`metric(self, embedding_a, embedding_b)`

Similarity between two embeddings.

Embeddings are of broadcastable shapes. (1 or batch_size)

Parameters:

Name	Type	Description	Default
`embedding_a`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*
`embedding_b`	`ndarray`	Embedding of shape (1 or batch_size, embedding_size)	*required*

Returns:

Type	Description
`ndarray`	Vector of distances (batch_size or 1,)

Source code in `npc_engine/models/similarity/similarity_transformers.py`

```
def metric(self, embedding_a: np.ndarray, embedding_b: np.ndarray) -> np.ndarray:
    """Similarity between two embeddings.

    Embeddings are of broadcastable shapes. (1 or batch_size)
    Args:
        embedding_a: Embedding of shape (1 or batch_size, embedding_size)
        embedding_b: Embedding of shape (1 or batch_size, embedding_size)

    Returns:
        Vector of distances (batch_size or 1,)
    """
    if self.metric_type == "dot":
        return -np.dot(embedding_a, embedding_b.T).squeeze(0)
    elif self.metric_type == "cosine":
        return 1 - cdist(embedding_a, embedding_b, metric="cosine").squeeze(0)
```

`stt` special

Speech to text API.

This module implements specific models and wraps them under the common interface for loading and inference.

Examples:

```
from npc_engine.models.stt import SpeechToTextAPI
model = SpeechToTextAPI.load("path/to/model_dir")
text = model.listen()  # Say something
```

`nemo_stt`

Module that implements Huggingface transformers semantic similarity.

`NemoSTT (SpeechToTextAPI)`

Text to speech pipeline based on Nemo toolkit.

Uses:

```
- ONNX export of EncDecCTCModel from Nemo toolkit.
- Punctuation distillbert model from Nemo toolkit. (requires tokenizer.json as well)
- Huggingface transformers model for predicting that sentence is finished
    (Cropped sentence -> 0 label, finished sentence -> 1 label).
- OpenSLR Librispeech 3-gram model converted to lowercase https://www.openslr.org/11/
```

https://github.com/NVIDIA/NeMo https://catalog.ngc.nvidia.com/orgs/nvidia/models/quartznet15x5 https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/main/nlp/punctuation_and_capitalization.html

ctc.onnx spec:

```
- inputs:
    `audio_signal` mel spectogram of shape `(batch_size, 64, mel_sequence)`
- outputs:
    `tokens` of shape `(batch_size, token_sequence, logits)`
```

punctuation.onnx spec:

```
- inputs:
    `input_ids` mel spectogram of shape `(batch_size, sequence)`
    `attention_mask` mel spectogram of shape `(batch_size, sequence)`
- outputs:
    `punctuation` of shape `(batch_size, sequence, 4)`
    `capitalization` of shape `(batch_size, sequence, 2)`
```

`__init__(self, model_path, frame_size=1000, sample_rate=16000, predict_punctuation=False, alpha=0.5, beta=1, *args, **kwargs)` special

Create and load biencoder model for semantic similarity.

Parameters:

Name	Type	Description	Default
`model_path`	`str`	A path where model config and weights are	*required*
`metric`		distance to compute semantic similarity	*required*

Source code in `npc_engine/models/stt/nemo_stt.py`

```
def __init__(
    self,
    model_path: str,
    frame_size: int = 1000,
    sample_rate: int = 16000,
    predict_punctuation: bool = False,
    alpha: float = 0.5,
    beta: float = 1,
    *args,
    **kwargs,
):
    """Create and load biencoder model for semantic similarity.

    Args:
        model_path: A path where model config and weights are
        metric: distance to compute semantic similarity
    """
    super().__init__(
        sample_rate=sample_rate,
        model_path=model_path,
        frame_size=frame_size,
        *args,
        **kwargs,
    )
    sess_options = rt.SessionOptions()
    sess_options.graph_optimization_level = opt_level.ORT_ENABLE_BASIC
    provider = rt.get_available_providers()[0]
    logger.info(f"STT uses {provider} provider")

    self.stft_filterbanks = librosa.filters.mel(
        16000, 512, n_mels=64, fmin=0, fmax=8000
    )
    self.stft_window = librosa.filters.get_window("hann", 320, fftbins=False)
    self.mel_mean, self.mel_std = self._fixed_normalization()

    self.asr_model = rt.InferenceSession(
        path.join(model_path, "ctc.onnx"),
        providers=[provider],
        sess_options=sess_options,
    )

    self.predict_punctuation = predict_punctuation
    if self.predict_punctuation:
        self.punctuation = rt.InferenceSession(
            path.join(model_path, "punctuation.onnx"),
            providers=[provider],
            sess_options=sess_options,
        )
        self.tokenizer = Tokenizer.from_file(
            path.join(model_path, "tokenizer.json")
        )
    self.asr_vocab = " abcdefghijklmnopqrstuvwxyz'"
    self.asr_vocab = list(self.asr_vocab)
    self.asr_vocab.append("")

    self.punct_labels = "O,.?"
    self.capit_labels = "OU"

    self.decoder = build_ctcdecoder(
        self.asr_vocab,
        kenlm_model_path=path.join(model_path, "lowercase_3-gram.pruned.1e-7.arpa"),
        alpha=alpha,  # tuned on a val set
        beta=beta,  # tuned on a val set
    )

    sess_options = rt.SessionOptions()
    sess_options.graph_optimization_level = opt_level.ORT_ENABLE_ALL
    self.sentence_model = rt.InferenceSession(
        path.join(model_path, "sentence_prediction.onnx"),
        providers=[rt.get_available_providers()[0]],
        sess_options=sess_options,
    )
```

```
    self.sentence_tokenizer = Tokenizer.from_file(
        path.join(model_path, "sentence_tokenizer.json")
    )

    logger.info(
        f"Sentence classifier uses {rt.get_available_providers()[0]} provider"
    )
```

`decide_finished(self, context, text)`

Decide if audio transcription should be finished.

Parameters:

Name	Type	Description	Default
`context`	`str`	Text context of the speech recognized (e.g. a question to which speech recognized is a reply to).	*required*
`text`	`str`	Recognized speech so far	*required*
`pause_time`		Pause after last speech in milliseconds	*required*

Returns:

Type	Description
`bool`	Decision to stop recognition and finalize results.

Source code in `npc_engine/models/stt/nemo_stt.py`

```
def decide_finished(self, context: str, text: str) -> bool:
    """Decide if audio transcription should be finished.

    Args:
        context: Text context of the speech recognized
            (e.g. a question to which speech recognized is a reply to).
        text: Recognized speech so far
        pause_time: Pause after last speech in milliseconds

    Returns:
        Decision to stop recognition and finalize results.
    """
    decision = self._decide_sentence_finished(context, text)
    print("Sentence decision:", decision)
    done = bool(decision)
    return done
```

`decode(self, logits)`

Decode logits into text.

Parameters:

Name	Type	Description	Default
`logits`	`ndarray`	ndarray of float32 of shape (timesteps, vocab_size).	*required*

Returns:

Type	Description
`str`	Decoded string.

Source code in `npc_engine/models/stt/nemo_stt.py`

```
def decode(self, logits: np.ndarray) -> str:
    """Decode logits into text.

    Args:
        logits: ndarray of float32 of shape (timesteps, vocab_size).

    Returns:
        Decoded string.
    """
    return self.decoder.decode(logits)
```

`postprocess(self, text)`

Add punctuation and capitalization.

Parameters:

Name	Type	Description	Default
`text`	`str`	audio transcription.	*required*

Returns:

Type	Description
`str`	Postprocessed text transcribtion.

Source code in `npc_engine/models/stt/nemo_stt.py`

```
def postprocess(self, text: str) -> str:
    """Add punctuation and capitalization.

    Args:
        text: audio transcription.

    Returns:
        Postprocessed text transcribtion.
    """
    if self.predict_punctuation:
        enc = self.tokenizer.encode(text)

        punct, capit = self.punctuation.run(
            None,
            {
                "input_ids": np.asarray(enc.ids, np.int64).reshape([1, -1]),
                "attention_mask": np.asarray(enc.attention_mask, np.int64).reshape(
                    [1, -1]
                ),
            },
        )
        punct = punct.argmax(2)
        capit = capit.argmax(2)
        punctuated_capitalized = self._apply_punct_capit_predictions(
            text, punct[0][1:-1].tolist(), capit[0][1:-1].tolist()
        )
        return punctuated_capitalized
    else:
        return text
```

`transcribe(self, audio)`

Transcribe audio usign this pipeline.

Parameters:

Name	Type	Description	Default
`audio`	`List[float]`	ndarray of int16 of shape (samples,).	*required*

Returns:

Type	Description
`ndarray`	Transcribed text from the audio.

Source code in `npc_engine/models/stt/nemo_stt.py`

```
def transcribe(self, audio: List[float]) -> np.ndarray:
    """Transcribe audio usign this pipeline.

    Args:
        audio: ndarray of int16 of shape (samples,).

    Returns:
        Transcribed text from the audio.
    """
    logits = self._predict(np.asarray(audio, dtype=np.float32))
    return logits
```

stt_base

Module that implements speech to text model API.

SpeechToTextAPI (Model)

Abstract base class for speech to text models.

`__del__(self)` special

Stop listening on destruction.

Source code in `npc_engine/models/stt/stt_base.py`

```
def __del__(self):
    """Stop listening on destruction."""
    if self.microphone_initialized:
        self.stream.stop()
```

`__init__(self, min_speech_duration=100, max_silence_duration=1000, vad_mode=None, sample_rate=16000, vad_frame_ms=10, pad_size=1000, *args, **kwargs)` special

Initialize VAD part of the API.

Source code in `npc_engine/models/stt/stt_base.py`

```
def __init__(
    self,
    min_speech_duration=100,
    max_silence_duration=1000,
    vad_mode=None,
    sample_rate=16000,
    vad_frame_ms=10,
    pad_size=1000,
    *args,
    **kwargs,
):
    """Initialize VAD part of the API."""
    super().__init__()
    self.initialized = True
    sd.default.samplerate = sample_rate

    self.max_silence_duration = max_silence_duration
    self.min_speech_duration = min_speech_duration
    self.vad = webrtcvad.Vad()
    if vad_mode is not None:
        self.vad.set_mode(vad_mode)
    self.sample_rate = sample_rate
    self.listen_queue = Queue(10)
    self.vad_frame_ms = vad_frame_ms
    self.vad_frame_size = int((vad_frame_ms * sample_rate) / 1000)
    self.running = False
    self.microphone_initialized = False
    self.silence_buffer = np.empty([0])
    self.pad_size = pad_size
```

`decide_finished(self, context, text)`

Abstract method for deciding if audio transcription should be finished.

Should be implemented by the specific model.

Parameters:

Name	Type	Description	Default
`context`	`str`	Text context of the speech recognized (e.g. a question to which speech recognized is a reply to).	*required*
`text`	`str`	Recognized speech so far.	*required*

Returns:

Type	Description
`bool`	Decision to stop recognition and finalize results.

Source code in `npc_engine/models/stt/stt_base.py`

```
@abstractmethod
def decide_finished(self, context: str, text: str) -> bool:
    """Abstract method for deciding if audio transcription should be finished.

    Should be implemented by the specific model.

    Args:
        context: Text context of the speech recognized
            (e.g. a question to which speech recognized is a reply to).
        text: Recognized speech so far.

    Returns:
        Decision to stop recognition and finalize results.
    """
    return None
```

`decode(self, logits)`

Decode logits into text.

Parameters:

Name	Type	Description	Default
`logits`	`ndarray`	ndarray of float32 of shape (timesteps, vocab_size).	*required*

Returns:

Type	Description
`str`	Decoded string.

Source code in `npc_engine/models/stt/stt_base.py`

```
@abstractmethod
def decode(self, logits: np.ndarray) -> str:
    """Decode logits into text.

    Args:
        logits: ndarray of float32 of shape (timesteps, vocab_size).

    Returns:
        Decoded string.
    """
    return None
```

`get_devices(self)`

Get available audio devices.

Source code in `npc_engine/models/stt/stt_base.py`

```
def get_devices(self) -> Dict[int, str]:  # pragma: no cover
    """Get available audio devices."""
    return [device["name"] for device in sd.query_devices()]
```

`initialize_microphone_input(self)`

Initialize microphone.

Source code in `npc_engine/models/stt/stt_base.py`

```
def initialize_microphone_input(self):
    """Initialize microphone."""
    if self.microphone_initialized:
        return
    self.running = False
    self.microphone_initialized = True

    def callback(in_data, frame_count, time_info, status):
        if self.running:
            try:
                self.listen_queue.put(in_data.reshape(-1), block=False)
            except Exception:
                return
        else:
            if not self._vad_frame(
                in_data.reshape(-1)
            ):  # Register only silence for buffer
                self.silence_buffer = np.append(
                    self.silence_buffer, in_data.reshape(-1)
                )
                self.silence_buffer = self.silence_buffer[-self.pad_size :]

    self.stream = sd.InputStream(
        samplerate=self.sample_rate,
        channels=1,
        blocksize=self.vad_frame_size,
        callback=callback,
    )
    self.stream.start()
    while self.silence_buffer.shape[0] < self.pad_size:
        pass
```

`listen(self, context=None)`

Listen for speech input and return text from speech when done.

Listens for speech, if speech is active for longer than self.frame_size in milliseconds then starts transcribing it. On each voice activity detection (VAD) pause uses context to decide if transcribed text is a finished response to a context. If it is, applies preprocessing and returns the result. If transcribed text is not a response to a context but VAD pause persists through max_silence_duration then returns the results anyway.

Requires a microphone input to be initialized.

Parameters:

Name	Type	Description	Default
`context`	`str`	A last line of the dialogue used to decide when to stop listening. It allows our STT system to not wait for a VAD timeout (max_silence_duration in ms).	`None`

Returns:

Type	Description
`str`	Recognized text from the audio.

Source code in `npc_engine/models/stt/stt_base.py`

```
def listen(self, context: str = None) -> str:  # pragma: no cover
    """Listen for speech input and return text from speech when done.

    Listens for speech, if speech is active for longer than self.frame_size in milliseconds
    then starts transcribing it. On each voice activity detection (VAD) pause
    uses context to decide if transcribed text is a finished response to a context.
    If it is, applies preprocessing and returns the result.
    If transcribed text is not a response to a context but VAD pause persists through max_silence_duration
    then returns the results anyway.

    Requires a microphone input to be initialized.

    Args:
        context: A last line of the dialogue used to decide when to stop listening.
            It allows our STT system to not wait for a VAD timeout (max_silence_duration in ms).

    Returns:
        Recognized text from the audio.
    """
    if not self.microphone_initialized:
        raise RuntimeError("Microphone not initialized.")
    self.listen_queue.queue.clear()
    context = re.sub(r"[^A-Za-z0-9 ]+", "", context).lower() if context else None
    text = self._transcribe_vad_pause(context)
    processed = self.postprocess(text)

    self.listen_queue.queue.clear()
    return processed
```

`postprocess(self, text)`

Abstract method for audio transcription postprocessing.

Should be implemented by the specific model.

Parameters:

Name	Type	Description	Default
`text`	`str`	audio transcription.	*required*

Returns:

Type	Description
`str`	Postprocessed text transcribtion.

Source code in `npc_engine/models/stt/stt_base.py`

```
@abstractmethod
def postprocess(self, text: str) -> str:
    """Abstract method for audio transcription postprocessing.

    Should be implemented by the specific model.

    Args:
        text: audio transcription.

    Returns:
        Postprocessed text transcribtion.
    """
    return None
```

`select_device(self, device_id)`

Get available audio devices.

Source code in `npc_engine/models/stt/stt_base.py`

```
def select_device(self, device_id: int):  # pragma: no cover
    """Get available audio devices."""
    device_id = int(device_id)
    if device_id >= len(sd.query_devices()) or device_id < 0:
        raise ValueError(
            f"Bad device id, valid device ids in range [0;{len(sd.query_devices())})"
        )
    sd.default.device = device_id
```

`stt(self, audio)`

Transcribe speech.

Parameters:

Name	Type	Description	Default
`audio`	`List[int]`	PMC data with bit depth 16.	*required*

Returns:

Type	Description
`str`	Recognized text from the audio.

Source code in `npc_engine/models/stt/stt_base.py`

```
def stt(self, audio: List[int]) -> str:
    """Transcribe speech.

    Args:
        audio: PMC data with bit depth 16.

    Returns:
        Recognized text from the audio.
    """
    logits = self.transcribe(audio)
    text = self.decode(logits)
    text = self.postprocess(text)
    return text
```

`transcribe(self, audio)`

Abstract method for audio transcription.

Should be implemented by the specific model.

Parameters:

Name	Type	Description	Default
`audio`	`ndarray`	ndarray of int16 of shape (samples,).	*required*

Returns:

Type	Description
`ndarray`	Transcribed logits from the audio.

Source code in `npc_engine/models/stt/stt_base.py`

```
@abstractmethod
def transcribe(self, audio: np.ndarray) -> np.ndarray:
    """Abstract method for audio transcription.

    Should be implemented by the specific model.

    Args:
        audio: ndarray of int16 of shape (samples,).

    Returns:
        Transcribed logits from the audio.
    """
    return None
```

`tts` special

Text to speech specific model implementations.

This module implements specific models and wraps them under the common interface for loading and inference.

Examples:

```
from npc_engine.models.tts import TextToSpeechAPI
model = TextToSpeechAPI.load("path/to/model_dir")
model.run(speaker_id=0, text="Hello, world!")
```

`flowtron`

Flowtron (https://github.com/NVIDIA/flowtron) text to speech inference implementation.

`FlowtronTTS (TextToSpeechAPI)`

Implements Flowtron architecture inference.

Paper:
arXiv:2005.05957
Code:
https://github.com/NVIDIA/flowtron

Onnx export script can be found in this fork https://github.com/npc-engine/flowtron.

This model class requires four ONNX models `encoder.onnx`, `backward_flow.onnx`, `forward_flow.onnx` and `vocoder.onnx` where first three are layers from Flowtron architecture (`flow` corresponding to one direction pass of affine coupling layers) and `vocoder.onnx` is neural vocoder.

For detailed specs refer to https://github.com/npc-engine/flowtron.

`__init__(self, model_path, max_frames=400, gate_threshold=0.5, sigma=0.8, smoothing_window=3, smoothing_weight=0.5, *args, **kwargs)` special

Create and load Flowtron and vocoder models.

Source code in `npc_engine/models/tts/flowtron.py`

```
def __init__(
    self,
    model_path,
    max_frames=400,
    gate_threshold=0.5,
    sigma=0.8,
    smoothing_window=3,
    smoothing_weight=0.5,
    *args,
    **kwargs
):
    """Create and load Flowtron and vocoder models."""
    super().__init__(*args, **kwargs)
    sess_options = onnxruntime.SessionOptions()
    sess_options.graph_optimization_level = (
        onnxruntime.GraphOptimizationLevel.ORT_ENABLE_ALL
    )
    provider = onnxruntime.get_available_providers()[0]
    logging.info("FlowtronTTS using provider {}".format(provider))
    self.max_frames = max_frames
    self.gate_threshold = gate_threshold
    self.sigma = sigma
    self.smoothing_window = smoothing_window
    self.smoothing_weight = smoothing_weight

    self.encoder = onnxruntime.InferenceSession(
        path.join(model_path, "encoder.onnx"),
        providers=[provider],
        sess_options=sess_options,
    )
    self.backward_flow = onnxruntime.InferenceSession(
        path.join(model_path, "backward_flow.onnx"),
        providers=[provider],
        sess_options=sess_options,
    )
    self.forward_flow = onnxruntime.InferenceSession(
        path.join(model_path, "forward_flow.onnx"),
        providers=[provider],
        sess_options=sess_options,
    )
    self.vocoder = onnxruntime.InferenceSession(
        path.join(model_path, "vocoder.onnx"),
        providers=[provider],
        sess_options=sess_options,
    )
    self.speaker_ids = [str(i) for i in range(127)]
    self.speaker_ids_map = {idx: i for i, idx in enumerate(self.speaker_ids)}
```

`get_speaker_ids(self)`

Return available ids of different speakers.

Source code in `npc_engine/models/tts/flowtron.py`

```
def get_speaker_ids(self) -> List[str]:
    """Return available ids of different speakers."""
    return self.speaker_ids
```

`run(self, speaker_id, text, n_chunks)`

Create a generator for iterative generation of speech.

Parameters:

Name	Type	Description	Default
`speaker_id`	`str`	Id of the speaker.	*required*
`text`	`str`	Text to generate speech from.	*required*
`n_chunks`	`int`	Number of chunks to split generation into.	*required*

Returns:

Type	Description
`Iterator[numpy.ndarray]`	Generator that yields next chunk of speech in the form of f32 ndarray.

Source code in `npc_engine/models/tts/flowtron.py`

```
def run(self, speaker_id: str, text: str, n_chunks: int) -> Iterator[np.ndarray]:
    """Create a generator for iterative generation of speech.

    Args:
        speaker_id: Id of the speaker.
        text: Text to generate speech from.
        n_chunks: Number of chunks to split generation into.

    Returns:
        Generator that yields next chunk of speech in the form of f32 ndarray.
    """
    print(text)
    text = self._get_text(text)
    speaker_id = np.asarray([[self.speaker_ids_map[speaker_id]]], dtype=np.int64)
    enc_outps_ortvalue = onnxruntime.OrtValue.ortvalue_from_shape_and_type(
        [text.shape[1], 1, 640], np.float32, "cpu", 0
    )

    io_binding = self.encoder.io_binding()
    io_binding.bind_ortvalue_output("text_emb", enc_outps_ortvalue)
    io_binding.bind_cpu_input("speaker_vecs", speaker_id)
    io_binding.bind_cpu_input("text", text.reshape([1, -1]))
    self.encoder.run_with_iobinding(io_binding)

    residual = np.random.normal(
        0, self.sigma, size=[self.max_frames, 1, 80]
    ).astype(np.float32)

    residual = self._run_backward_flow(residual, enc_outps_ortvalue)
    residual = self._run_forward_flow(
        residual, enc_outps_ortvalue, num_split=self.max_frames // n_chunks
    )
    last_audio = None
    for residual in residual:
        residual = np.transpose(residual, axes=(1, 2, 0))
        audio = self.vocoder.run(None, {"mels": residual})[0]
        # audio = np.where(
        #     (audio > (audio.mean() - audio.std()))
        #     | (audio < (audio.mean() + audio.std())),
        #     audio,
        #     audio.mean(),
        # )
        tmp = audio
        if last_audio is None:
            audio = audio[:, 1000:]
        if last_audio is not None:
            cumsum_vec = np.cumsum(
                np.concatenate([last_audio, audio], axis=1), axis=1
            )
            ma_vec = (
                cumsum_vec[:, self.smoothing_window :]
                - cumsum_vec[:, : -self.smoothing_window]
            ) / self.smoothing_window
            audio = (1 - self.smoothing_weight) * audio[
                :, self.smoothing_window :
            ] + self.smoothing_weight * ma_vec[:, last_audio.shape[1] :]
        last_audio = tmp
        audio = audio.reshape(-1)
        # audio = audio / np.abs(audio).max()
        yield audio
```

tts_base

Module that implements text to speech model API.

TextToSpeechAPI (Model)

Abstract base class for text-to-speech models.

__init__(self, *args, **kwargs) special

Empty initialization method for API to be similar to other model base classes.

Source code in npc_engine/models/tts/tts_base.py

```
def __init__(self, *args, **kwargs) -> None:
    """Empty initialization method for API to be similar to other model base classes."""
    self.generator = None
    super().__init__()
    self.initialized = True
```

get_speaker_ids(self)

Get list of available speaker ids.

Returns:

Type	Description
List[str]	The return value. True for success, False otherwise.

Source code in npc_engine/models/tts/tts_base.py

```
@abstractmethod
def get_speaker_ids(self) -> List[str]:
    """Get list of available speaker ids.

    Returns:
        The return value. True for success, False otherwise.
    """
    return None
```

run(self, speaker_id, text, n_chunks)

Create a generator for iterative generation of speech.

Parameters:

Name	Type	Description	Default
speaker_id	str	Id of the speaker.	*required*
text	str	Text to generate speech from.	*required*
n_chunks	int	Number of chunks to split generation into.	*required*

Returns:

Type	Description
`Iterable[numpy.ndarray]`	Generator that yields next chunk of speech in the form of f32 ndarray.

Source code in `npc_engine/models/tts/tts_base.py`

```
@abstractmethod
def run(self, speaker_id: str, text: str, n_chunks: int) -> Iterable[np.ndarray]:
    """Create a generator for iterative generation of speech.

    Args:
        speaker_id: Id of the speaker.
        text: Text to generate speech from.
        n_chunks: Number of chunks to split generation into.

    Returns:
        Generator that yields next chunk of speech in the form of f32 ndarray.
    """
    return None
```

`tts_get_results(self)`

Retrieve the next chunk of generated speech.

Returns:

Type	Description
`Iterable[numpy.ndarray]`	Next chunk of speech in the form of f32 ndarray.

Source code in `npc_engine/models/tts/tts_base.py`

```
def tts_get_results(self) -> Iterable[np.ndarray]:
    """Retrieve the next chunk of generated speech.

    Returns:
        Next chunk of speech in the form of f32 ndarray.
    """
    if self.generator is not None:
        return next(self.generator).tolist()
    else:
        raise ValueError(
            "Speech generation was not started. Use tts_start to start it"
        )
```

`tts_start(self, speaker_id, text, n_chunks)`

Initiate iterative generation of speech.

Parameters:

Name	Type	Description	Default
`speaker_id`	`str`	Id of the speaker.	*required*
`text`	`str`	Text to generate speech from.	*required*
`n_chunks`	`int`	Number of chunks to split generation into.	*required*

Source code in `npc_engine/models/tts/tts_base.py`

```
def tts_start(self, speaker_id: str, text: str, n_chunks: int) -> None:
    """Initiate iterative generation of speech.

    Args:
        speaker_id: Id of the speaker.
        text: Text to generate speech from.
        n_chunks: Number of chunks to split generation into.

    """
    sentences = re.split(r"(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?)\s", text)
    self.generator = self._chain_run(speaker_id, sentences, n_chunks)
```

`utils` special

Utility package.

`lru_cache`

LRU cache.

`NumpyLRUCache`

Dict based LRU cache for numpy arrays.

`__init__(self, size)` special

Crate cache.

Source code in `npc_engine/models/utils/lru_cache.py`

```
def __init__(self, size):
    """Crate cache."""
    self.size = size
    self.lru_cache = collections.OrderedDict()
    self.common_dim = None
```

`cache_compute(self, keys, function)`

Get batch from cache and compute missing.

Parameters:

Name	Type	Description	Default
`keys`	`List[Any]`	List of keys	*required*

Returns:

Type	Description
`np.ndarray or None`	Found entries concatenated over 0 axis. list(_) or None: Keys that were not found.

Source code in `npc_engine/models/utils/lru_cache.py`

```
def cache_compute(
    self, keys: List[Any], function: Callable
) -> Tuple[np.ndarray, List[Any]]:
    """Get batch from cache and compute missing.

    Args:
        keys: List of keys

    Returns:
        np.ndarray or None: Found entries concatenated over 0 axis.
        list(_) or None: Keys that were not found.
    """
    if len(self.lru_cache) == 0:
        result = function(keys)
        self.put_batch(keys, result)
        return result
    else:
        result = np.zeros((len(keys), *self.common_dim))
        items = [self._get(key) for key in keys]
        not_found = [key for item, key in zip(items, keys) if item is None]
        if len(not_found) > 0:
            computed = function(not_found)
        computed_idx = 0
        for idx, item in enumerate(items):
            if item is None:
                result_slc = tuple([idx] + [slice(None)] * len(self.common_dim))
                computed_slc = tuple(
                    [computed_idx] + [slice(None)] * len(self.common_dim)
                )
                result[result_slc] = computed[computed_slc]
                computed_idx += 1
            else:
                result_slc = tuple([idx] + [slice(None)] * len(self.common_dim))
                result[result_slc] = item
    return result
```

`put_batch(self, keys, values)`

Put batch to cache.

Parameters:

Name	Type	Description	Default
`keys`	`List[Any]`	List of keys	*required*
`values`	`ndarray`	Ndarray of shape (len(keys), *common_dim)	*required*

Source code in `npc_engine/models/utils/lru_cache.py`

```
def put_batch(self, keys: List[Any], values: np.ndarray):
    """Put batch to cache.
```

```
    Args:
        keys: List of keys
        values: Ndarray of shape (len(keys), *common_dim)
    """
    self._validate_shape(values)
    for key, item in zip(keys, values):
        self._put(key, item)
```

2.6.3 `text` special

from https://github.com/keithito/tacotron

`cleaners`

adapted from https://github.com/keithito/tacotron.

Cleaners are transformations that run over the input text at both training and eval time.

Cleaners can be selected by passing a comma-delimited list of cleaner names as the "cleaners" hyperparameter. Some cleaners are English-specific. You'll typically want to use: 1. "english_cleaners" for English text 2. "transliteration_cleaners" for non-English text that can be transliterated to ASCII using the Unidecode library (https://pypi.python.org/pypi/Unidecode) 3. "basic_cleaners" if you do not want to transliterate (in this case, you should also update the symbols in symbols.py to match your data).

`flowtron_cleaners(text)`

Clean text with a set of cleaners.

Source code in `npc_engine/text/cleaners.py`

```
def flowtron_cleaners(text):
    """Clean text with a set of cleaners."""
    text = collapse_whitespace(text)
    text = remove_hyphens(text)
    text = expand_datestime(text)
    text = expand_numbers(text)
    text = expand_safe_abbreviations(text)
    return text
```

`numbers`

from https://github.com/keithito/tacotron

`symbols`

from https://github.com/keithito/tacotron

2.6.4 `version`

This module contains project version information.

.. currentmodule:: npc_engine.version .. moduleauthor:: evil.unicorn1 evil.unicorn1@gmail.com

2.6.5 `zmq_server`

Module that implements ZMQ server communication over JSON-RPC 2.0 (https://www.jsonrpc.org/specification).

`ZMQServer`

Json rpc server over zmq.

`__init__(self, port)` *special*

Create a server on the port.

Source code in `npc_engine/zmq_server.py`

```
def __init__(self, port: str):
    """Create a server on the port."""
    print("starting server")
    self.context = zmq.Context()
    self.socket = self.context.socket(zmq.REP)
    self.socket.bind(f"tcp://*:{port}")
```

`run(self, api_dict)`

Run an npc-engine json rpc server and start listening.

Parameters:

Name	Type	Description	Default
`api_dict`		A Mapping from method names to callables that implement this method.	*required*

Source code in `npc_engine/zmq_server.py`

```
def run(self, api_dict):
    """Run an npc-engine json rpc server and start listening.

    Args:
        api_dict: A Mapping from method names to callables that implement this method.
    """
    dispatcher.update(api_dict)
    dispatcher.update({"status": lambda: "OK"})
    while True:
        message = self.socket.recv_string()
        logger.trace("Received request: %s" % message)

        start = time.time()
        response = JSONRPCResponseManager.handle(message, dispatcher)
        end = time.time()

        logger.info("Handle message time: %d" % (end - start))
        logger.trace("Message reply: %s" % (response.json))

        #  Send reply back to client
        self.socket.send_string(response.json)
```

3. Unity

3.1 Overview

This is the documentation for Unity integration for NPC Engine.

> Warning

Before using NPC Engine integration you must turn off play mode compilation in `Edit -> Preferences -> General -> Script changes while playing`. If play mode compilation will happen Unity will freeze and only way to restart it would be to kill the process manually!

3.1.1 Dependencies

- Welcome window depends on EditorCoroutines unity package.
  You can add this line to your Packages\manifest.json:

```
{
    "dependencies": {
        ...
        "com.unity.editorcoroutines": "1.0.0",
        ...
    }
}
```

- Advanced demo scene requires these free asset store packages:
  - VIDE dialogues
  - Modular First Person Controller
  - Low Poly Modular Armours
  - RPG Poly Pack - Lite

3.1.2 Getting started

NPC Engine is soon to be released on Asset Store, but for now:

- Clone Integration repository
- Install dependencies
- Move integration Assets folder to your Unity project.
- Follow welcome window instructions
- Check out Basic Demo tutorial to see the basic usage of the NPC-engine API
- Check out Advanced Demo to understand how higher-level components work and how to integrate NPC Engine into your game.

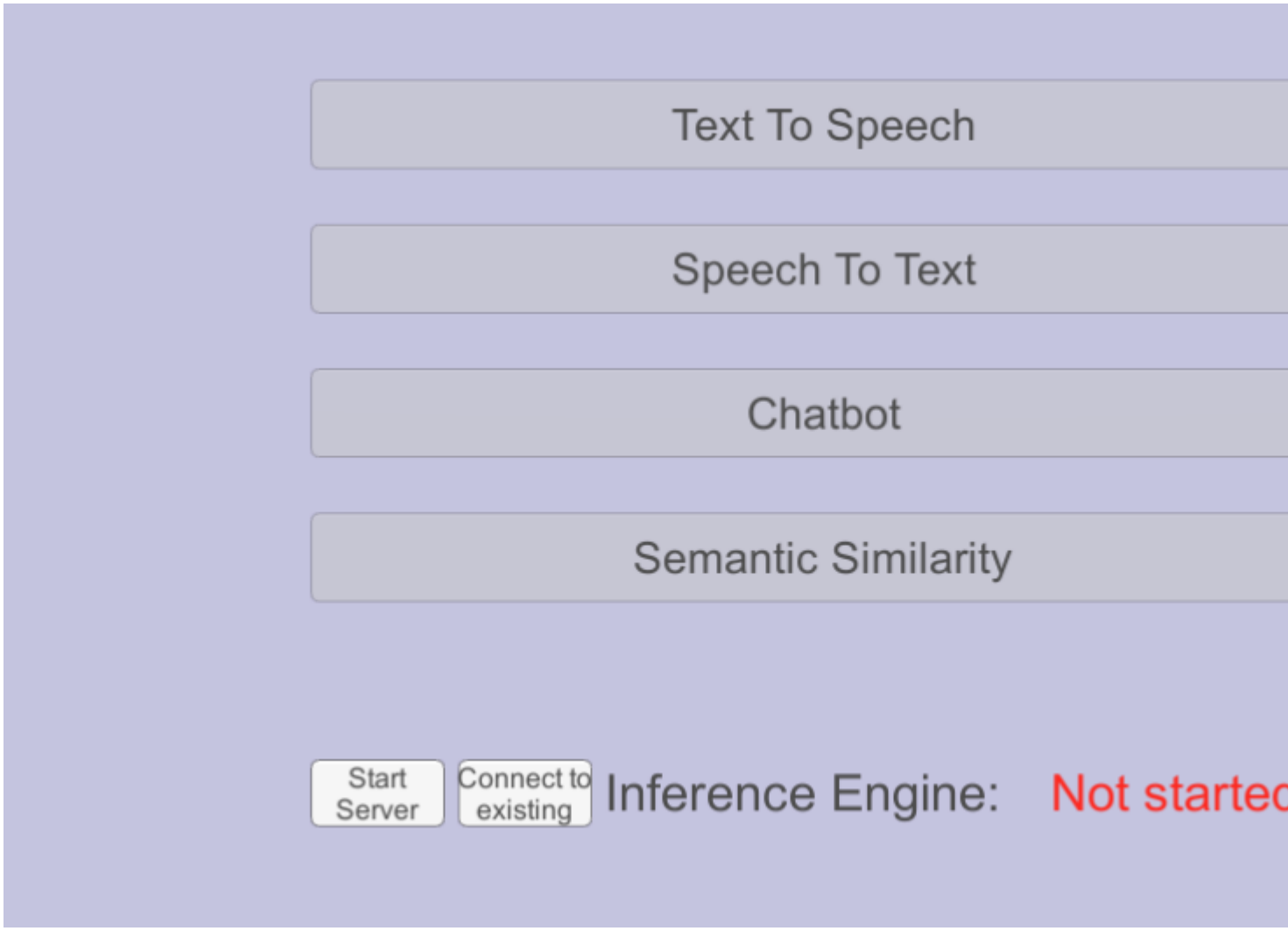
3.2 Basic Demo Tutorial

This tutorial explains raw usage of the NPC Engine API from Unity using Basic Demo scene.

3.2.1 Scene Overview

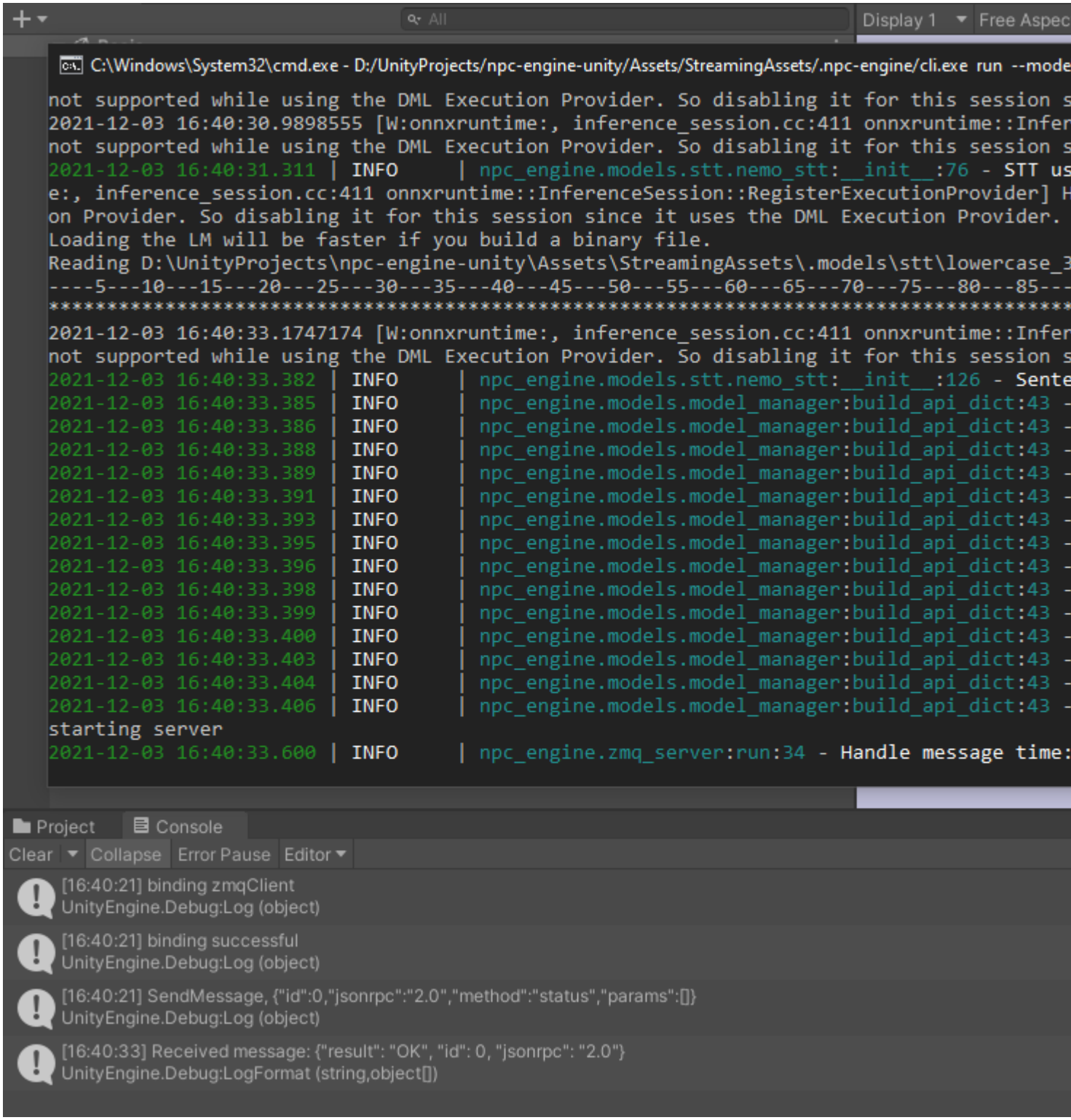
First lets go through and play around with the basic demo scene.
Its located under this path: `NPCEngine/Demo/BasicDemo/Basic.unity`

When you start it the first thing you'll see is this screen:



Since NPC Engine is a server that starts alongside unity and it's startup takes some time you can keep it running between playtests and just connect to existing one.
No server is running so you should start a new one.

When started you should see some Unity logs regarding connecting to the server as well as server console pop up with server logs. This behaviour is debug only and can be turned off by disabling `debug` flag in `NPCEngineManager` game object.

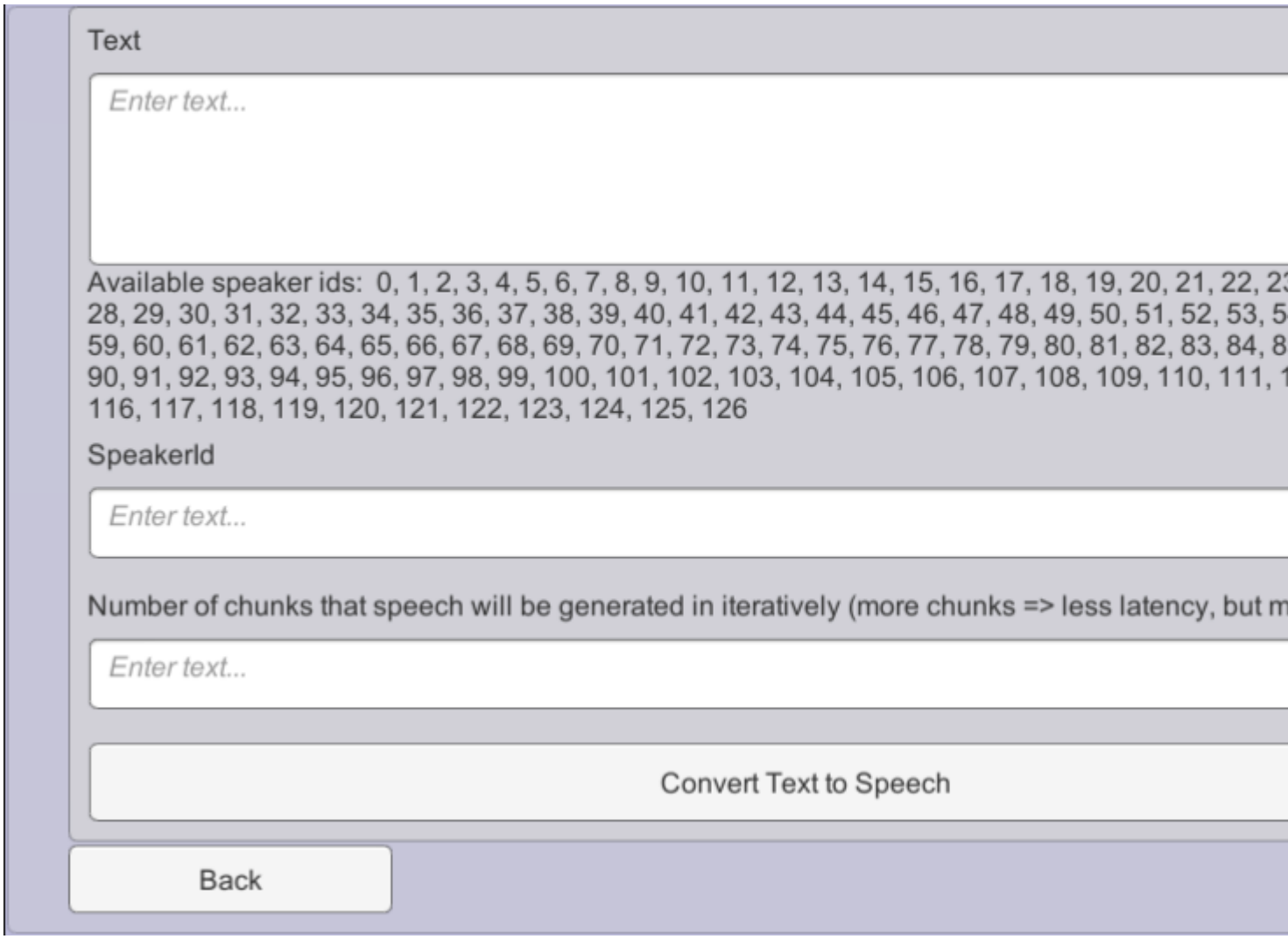


If NPC Engine starts successfully, menu options will become interactable and you will be able to play around with different APIs.

3.2.2 Available API Demos

Text To Speech Demo

This demo shows you the API that allows you to generate speech from text with multiple voices.



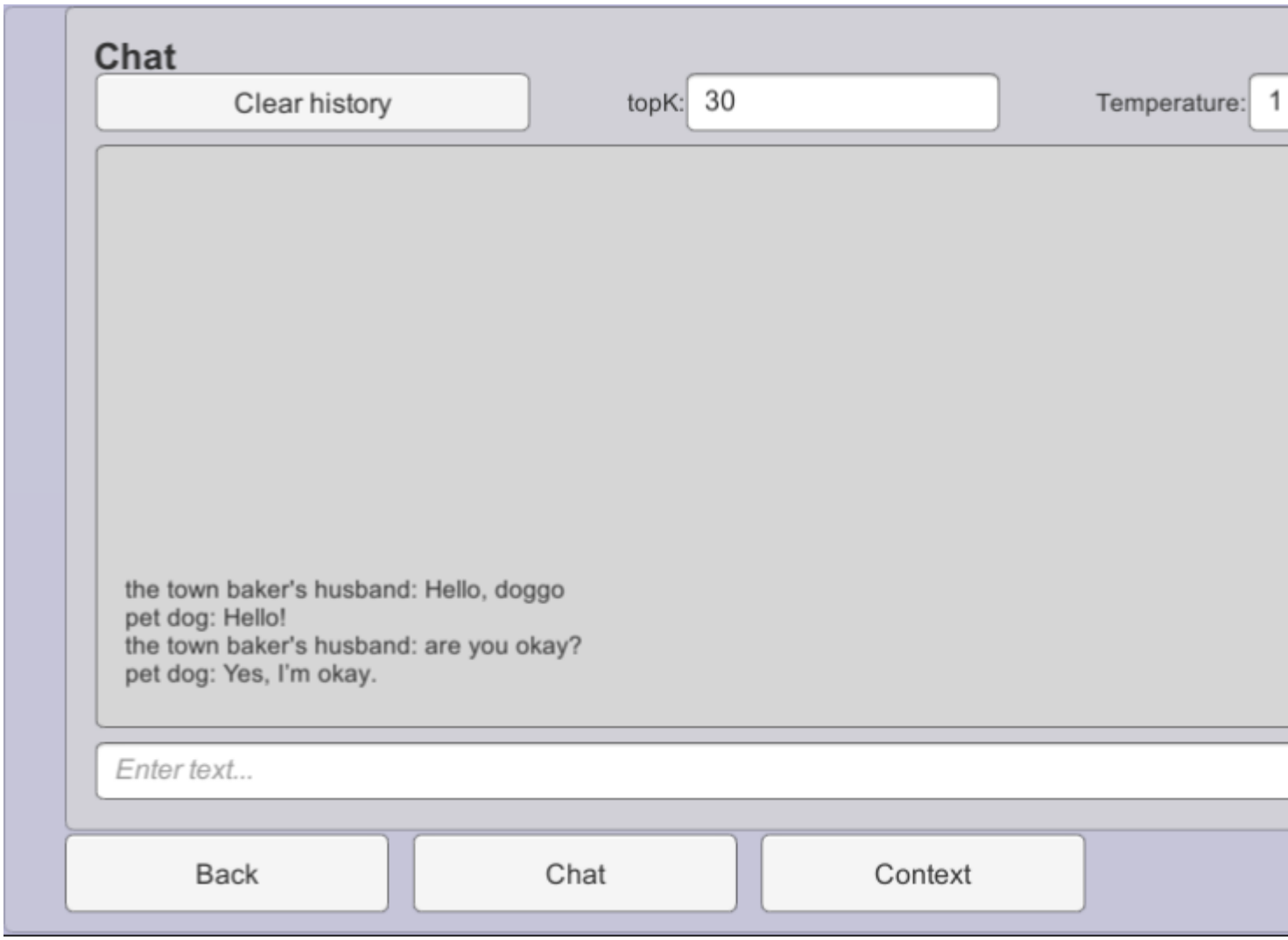
Fantasy Chatbot Demo

This demo shows you the chatbot API. It enables you to describe a fantasy character via the chatbot context and chat with your character.

Right now it's available only in the single style (Fantasy) but we are already working on the other chatbot neural networks with diffirent styles as well as tutorials how to train them yourself.

This demo greets you with a context in which you can fill in different descriptions to simulate different situations.

`Chat` button will take you to chat window where you can talk to the character defined in the context.

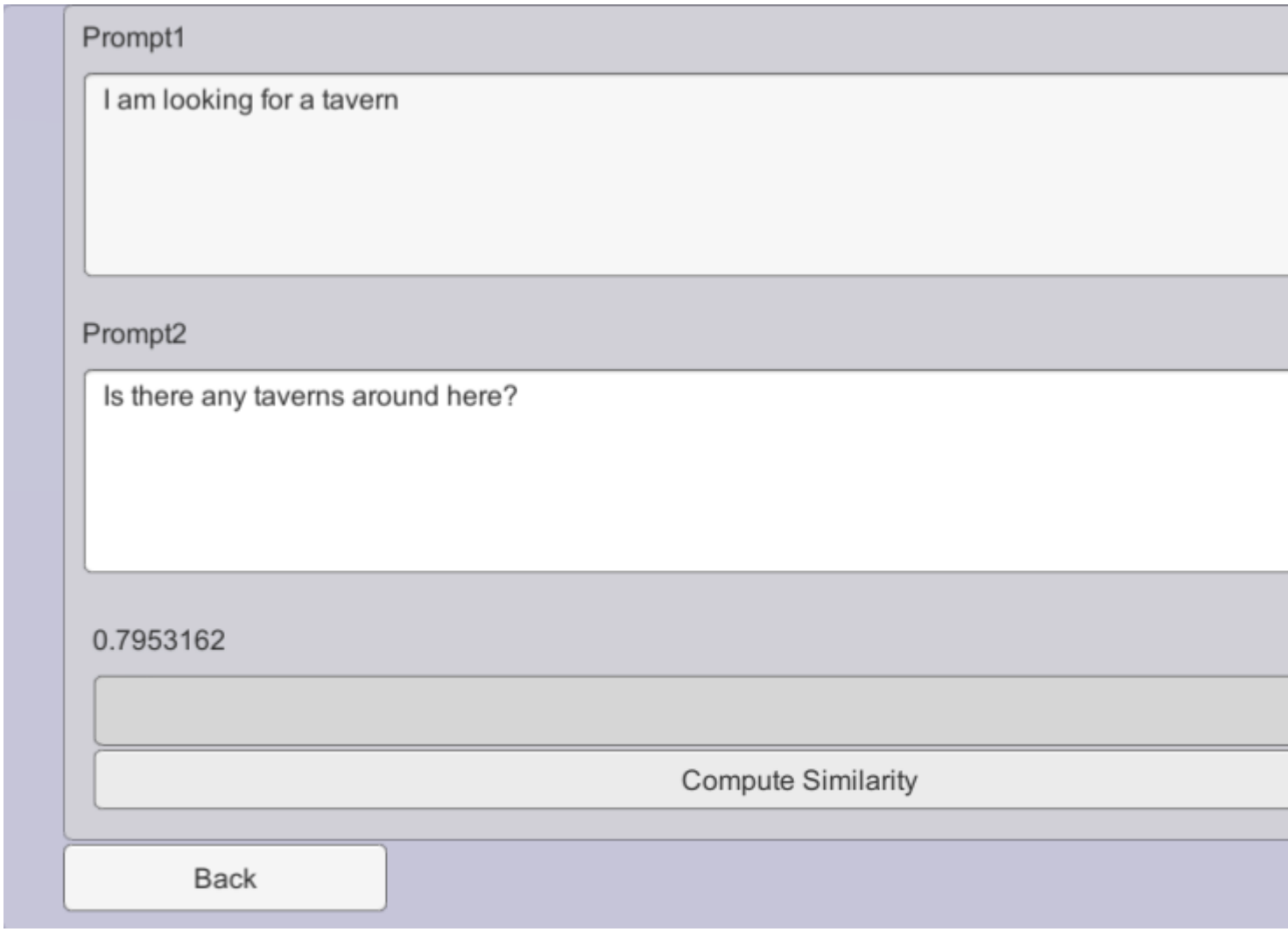


`Clear history` button will restart the dialogue.

Semantic Similarity Demo

This demo shows the API to compare two sentences via their meaning.
When you press `Compute Similarity` the score is shown in range of `[-1,1]`

Where -1 means that phrases are completely unrelated and 1 is that phrases are the same. Usually the most meaningful scores are in the range `[0,1]`



Speech To Text Demo

This demo shows you the API that allows you to listen to microphone input and transcribe it to text.

Just press `Listen` button and say something into the microphone.
Note that it will only work in low noise environment and with slow articulate speech.

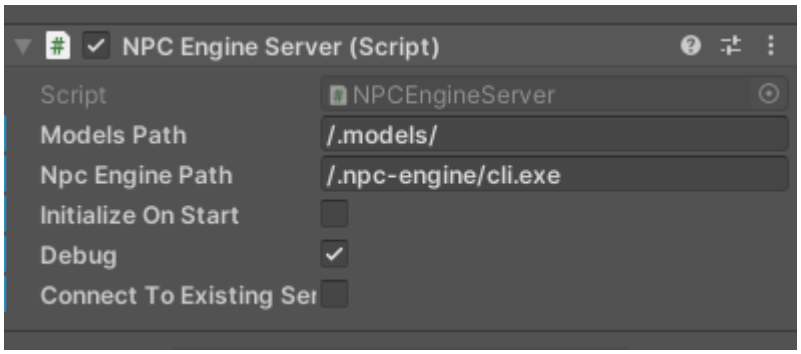
> Experimental API

This API is very WIP and experimental so it's performance is not yet ready for any production usage, you should use UnityEngine.Windows.Speech.DictationRecognizer instead

3.2.3 Server Lifetime

The main script that manages NPC Engine server is `NPCEngine.Server.NPCEngineServer`. It is attached to `NPCEngineManager` game object in the scene.

There you can find these public fields:



`Initialize On Start` controls whether NPCEngineServer will run `StartInferenceEngine` and `ConnectToServer` methods in it's `Awake` method. For the basic demo it's turned off to allow you to start and connect to server manually via UI buttons.

`Debug` flag when turned on, starts server in a CMD window as well as enables `NPCEngineServer` to write message logs to console. When it's off, server runs in the background with no logs produced.

`Connect To Existing Server` controls whether `NPCEngineServer` should start the server in `StartInferenceEngine` method and take ownership of the process (check it's health and terminate it `OnDestroy`). You can use this flag to not wait for NPCEngine to be initialized each playtest and keep connecting to the one that is already started.

3.2.4 API Deep Dive

Now as you have tried the functionality let's walk through the actual implementation.

Calling APIs

`NPCEngineServer` is a singleton and can be accessed via `NPCEngineServer.Instance` property. It also contains `NPCEngineServer.Initialized` property that turns to true if it was able to successfuly connect to the python process.

`NPCEngineServer` also implements `ResultFuture<R> Run<P, R>(String methodName, P parameters)` method that sends JSONRPC2.0 requests to the API, but it will throw an exception if NPCEngine was not initialized beforehand. This method returns `ResultFuture<R>` object that allows you to check if computation is finished and access deserialzed return type.

But generally there is no need to use this method directly, as there is already an implementation for each API in `NPCEngine.API` namespace.

Each API class implements all the required parameter and return types as well as API methods that return ResultFuture and their blocking coroutine versions.

Next sections will discuss each API, but to get more details you can refer to Models section of the Inference Engine docs.

`NPCENGINE.API.SEMANTICQUERY`

This static class exposes `Cache`, `Compare` and `CompareCoroutine` methods for similarity scoring. `Compare` and `CompareCoroutine` methods return similarity scores between query string and the batch of context strings.

`Cache` methods caches a batch of strings so that they can be compared to any other string at almost zero cost. Caching is performed via LRU cache and it's also done for every API input. Cache size can be controlled through semantic similarity config in models folder.

You can see the basic usage of this API in the `SemanticSimilarityCaller` script attached to `DemoUI/SemanticSimilarity` game object.

```
    ResultFuture<List<float>> result;

    private void Update()
    {
        if (result != null && result.ResultReady)
        {
            outputLabel.text = result.Result[0].ToString();
            result = null;
        }
    }

    public void CallSemanticSimilarity()
    {
        result = SemanticQuery.Compare(prompt1.text, new List<string> { prompt2.text });
    }
```

`NPCENGINE.API.CHATBOT<CHATBOTCONTEXT>`

The main difference you may notice is that API class is generic. This is because you have full control over what gets into the chatbot model as a prompt. Each chatbot npc-engine model has a jinja template in it's `config.yml` file. When chatbot API get's a request to generate text it uses this template to render string from context provided.

Default chatbot model context is implemented in `NPCEngine.Components.FantasyChatbotContext`.

Example API usage can be found in `ChatbotCaller` script.

All the other APIs follow the same patter as the two mentioned above. Refer to the corresponding caller to see the example usage.

To see the meaning of each of the API methods refer to Models section of the Inference Engine docs.

3.3 Advanced Demo Tutorial

This tutorial shows how to use NPC Engine higher level components as well as how to integrate them into the classic NPC design.

3.3.1 Overview

Dependencies

This scene depends on a these free packages:

- Modular First Person Controller is a player controller we are using. You can replace it with your own player controller including VR rigs. Custom Player Rig section explains how to do it.
- VIDE dialogues is a free dialogue tree implementation. This scene has an example integration for this dialogue system.
- Low Poly Modular Armours is used for character models.
- RPG Poly Pack - Lite is used for the scene itself.

Scene

This scene is located in `NPCEngine/Demo/AdvancedDemo` folder.

It contains 7 different characters with their own personas and names. Two of them have their own dialogue trees, two share the same dialogue tree and three do not have any dialogue trees assigned. Its a good example of how to use NPC Engine to fill the scene with NPCs.

To start the dialogue approach the character and start talking into your microphone.

> If you are using DictationRecognizerTTS (default option).

Dictation recognizer is currently functional only on Windows 10, and requires that dictation is permitted in the user's Speech privacy policy (Settings->Privacy->Speech, inking & typing). If dictation is not enabled, DictationRecognizerTTS will fail on Start. Developers can handle this failure in an app-specific way by providing a OnSpeechRecognitionFailed delegate.

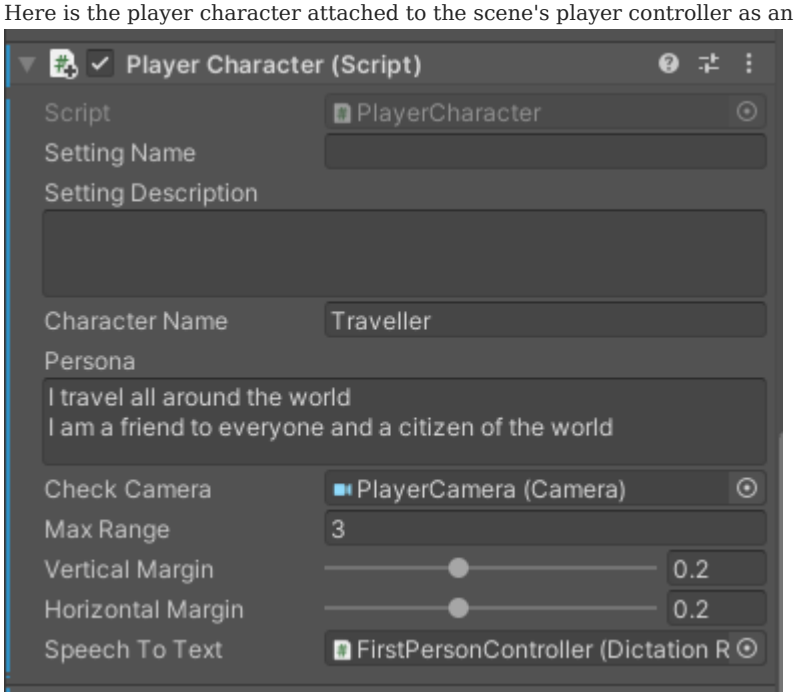
3.3.2 Components

Player Character

To integrate player controller into NPC Engine, you need to add two components to your player controller:

- `NPCEngine.Components.PlayerCharacter` : This is the main component that is responsible for the player's location, persona and ability to initiate dialogue.
  It should be attached to the gameobject that has `player` tagged collider so that `ColliderLocationTrigger` script works correctly. You should assign your player's camera to the `CheckCamera` field, It's used to check if player is looking at the NPC before initiating dialogue. You should also enter name and a persona of your player character. `MaxRange` is the minimum distance from the player to the NPC at which dialogue can be happening (dialogue is terminated if player is farther than this). `Vertical/HorizontalMargin` controls how centered should NPC be in the camera to initiate dialogue. Setting(Location) name and description are set by `ColliderLocationTrigger` script when player enters the location trigger.
- `NPCEngine.Components.AbstractSpeechToText` : This is the component that is responsible for the speech recognition. There are two implementations of this component available and they are discussed in the next section. By default, it's best to use `NPCEngine.Components.DictatinRecognizerSTT` which uses `UnityEngine.Windows.Speech.DictationRecognizer` and provides the best quality.

Here is the player character attached to the scene's player controller as an example:



Speech Recognition

There are two implementations of speech recognition available:

- `NPCEngine.Components.DictatinRecognizerSTT`
  It uses `UnityEngine.Windows.Speech.DictationRecognizer`.
  It's downside are:
  - It requires additional permissions to be enabled in the user's privacy settings.
  - It has relatively high latency.
  - It doesn't work when application is not in focus.
  - It's hard to diagnose if something goes wrong. (e.g. speech is not recognized)

But it does provide the best quality of recognition.

- `NPCEngine.Components.NPCEngineSTT`
  It uses NPCEngine's own speech recognition engine.
  It does not require additional permissions and has low latency, but it's work in progress and the quality is much worse than `NPCEngine.Components.DictatinRecognizerSTT`. It requires speech to be very clear and understandable as well as low noise environment. It also can be quite confusing for the chatbots when it does not recognize speech properly.

Advanced demo scene uses `DictatinRecognizerSTT` by default, but you can try `NPCEngineSTT` just by replacing components in `FirstPersonController` gameobject.

CollisionLocationTrigger

If your game has a lot of locations, you can use this component to make it easier to assign location names and descriptions to your player character. Just place a trigger collider to cover the location and add this component to it. Otherwise you could just provide default location name and description in the `PlayerCharacter` component.

Non-Player Character

To integrate NPC into NPC Engine, you need to:

- Implement `NPCEngine.Components.AbstractDialogueSystem`. It's already done for VIDE dialogue system in the demo scene in `NPCEngine/Demo/AdvancedDemo/Scripts` folder. Refer to VIDE Asset Store page for more details about this dialogue system.
- Add `NPCEngine.Components.AbstractDialogueSystem` and `NPCEngine.Components.NonPlayerCharacter` component to your NPC.

NPCENGINE.COMPONENTS.NONPLAYERCHARACTER

This component uses speech recognized by `PlayerCharacter` to navigate dialogue trees, generate replies and emit dialogue related events.

The high level flow of the dialogue is as follows:

- First, the type of the node is checked, if it's an NPC node, then the speech is generated and OnDialogueLine event is emitted. It repeats this process until a player node is found.
- When a player node is found, component signals PlayerCharacter to recognize more speech.
- When speech is recognized, OnDialogueLine event is emitted again for the player line, topics are requested from the dialogue system and OnTopicHintsUpdate is emitted.
- Player line is matched via semantic similarity to the player options in the dialogue tree.
- If the player line is matched to one of the options it is selected in the dialogue tree and AbstractDialogueSystem.Next() is called, otherwise reply is generated by the ChatbotAPI.
- All the steps are repeated until the dialogue is finished.

So as you can see you design your dialogue tree in the same way as you would without NPC Engine and everything else will be handled by the chatbot neural net.

The most important fields of this component should sound familiar for you if you've tried BasicDemo scene already. Here is the short description of those:

characterName and persona

These are the name and persona of the NPC used to generate lines via chatbot neural network.

topK and temperature

These are sampling parameters for the chatbot neural network. It was finetuned for the `temperature` == 1.0, so it's best to keep it that way. Randomness of the output can be controlled via `topK` parameter.

defaultThreshold

This is the default semantic similarity threshold that triggers dialogue options. You can also specify it in the dialogue system. In case of VIDE it can be added as extraVars to the dialogue node.

voiceId and nChunksTextGeneration

These are the parameters for TextToSpeech generation. VoiceId is the voice used to generate the text. nChunksTextGeneration is the number of chunks in which speech will be generated. In short, nChunksTextGeneration is a tradeof between quality and latency where 1 is the best quality and the most latency. Recommended range is [1, 10].

audioSourceQueue

It's a reference to the script that handles audio playback from the iterative speech generation.

dialogueSystem

It's a reference to the implementation of the AbstractDialogueSystem that is used to generate dialogue.

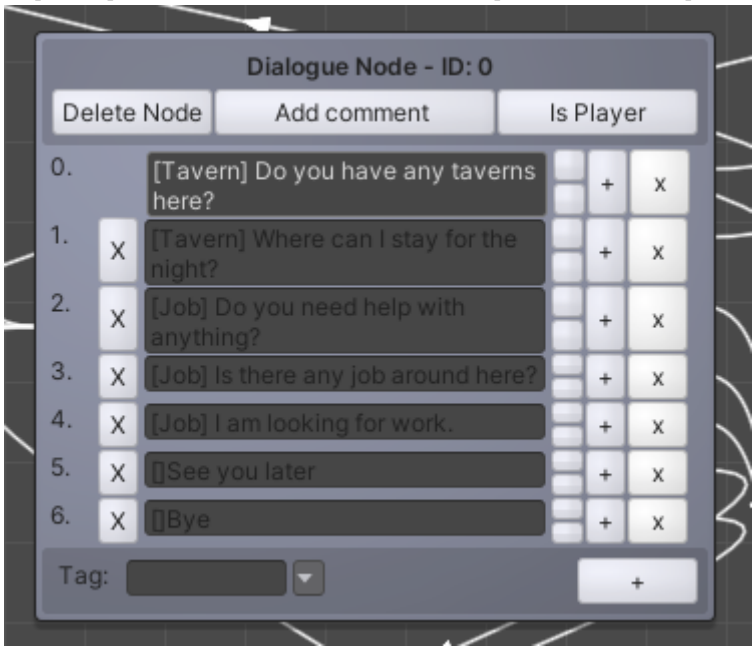
Events

They are pretty self-explanatory and are useful for all the presentation functionality (e.g. dialogue UI and animations).

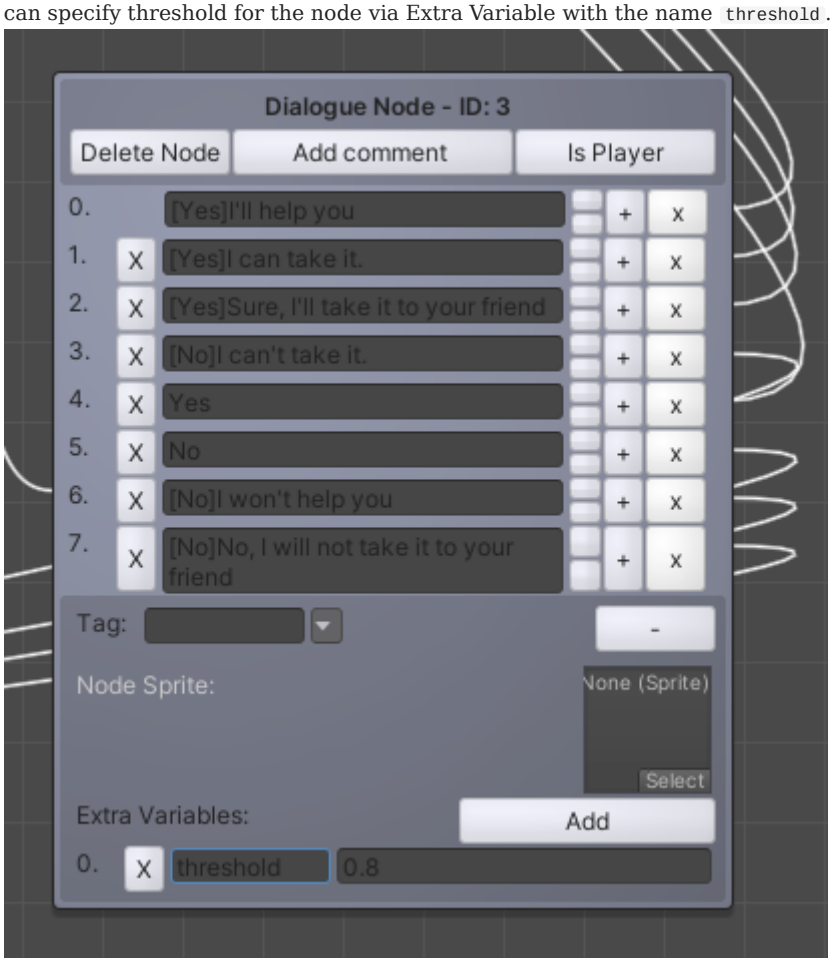
VIDEDIALOGUESYSTEM

This is the implementation of the AbstractDialogueSystem for the VIDE dialogues. Only a few things are different from the default usage of VIDE:

- You can specify topics in the line string by using square brackets in the beginning of the line (e.g. `[Tavern] Where is the tavern?`).
- OnTopicHintsUpdate event receives only unique topics as an argument so you can specify same topic for multiple lines
- If no topic is specified then the line is taken as a topic. You can set topic to empty via empty brackets.

e.g. 

- You can specify threshold for the node via Extra Variable with the name `threshold`.

e.g. 

Dialogue Design Considerations

You can check scene's existing dialogues for examples of how to design dialogues. Here are a few tips:

- Use multiple lines for each options to cover semantically distinct answers that in the context of the dialogue lead to similar results.

Example

In context of accepting to help someone do something there are a few options that are not semantically similar:

- `I will help you`
- `I will do something`
- `I'll figure something out`

Would all mean the same thing in the context of the dialogue, but in isolation mean different things. Best way to design dialogues for NPC Engine is to continually playtest them and find missing options that should be there as well as tune the thresholds to exclude anything unrelated.

- Start the dialogue via NPC node and use it to set the topic and the mood of the dialogue. It is the most reliable method to control what chatbot will generate.

Example

If the character is angry, then it's best to start the dialogue with NPC expressing this anger via cursing or complaining about the object of his anger.

If the character's village is attacked by goblins, then it's best to start the dialogue with a line that describes the situation and communicates distress.

4. Benchmarks

What about performance?

4.1 Here are the numbers:

4.1.1 i5-9600K + GTX1070 with default models

GPU VRAM

Before starting inference engine:
memory.used [MiB]
1213 MiB

After starting inference engine:
memory.used [MiB]
4310 MiB

Text to speech

Latency (time before first result): 1.0473401546478271 seconds
All the next iterations have real-time factor < 1.0

Semantic similarity

Similarity betwen short phrases 'I will help you' and 'I shall provide you my assistance' is computed in 0.06283211708068848s

Chatbot

Chatbot reply `Hello partner! Ornament please. Such a delightful pup! You are loyally loyal to your master?` to a big context generated in 1.411924123764038s

4.2 Run the benchmark test on your computer

> Warning

Requires Nvidia GPU because it uses nvidia-smi command line tool

- Comment out test skipping in `tests\benchmarks\benchmark.py`
- Run it with `pytest tests\benchmarks\benchmark.py -s`

At the moment the output is not pretty, it's a work in progress.