Kenenhrur Language Documentation

Welcome to the Kenenhrur documentation! This guide offers a comprehensive overview of the Kenenhrur language, its core libraries, and how to harness its unique capabilities for intelligent computer-generated speech and imagery.

1. Getting Started

1.1. Installation

Kenenhrur is currently under active development. For early access and installation instructions, please send an email to the Keeper: michael@ma8.company

1.2. Your First Kenenhrur Program: "Hello, World!" in Speech

Let's begin with a simple program that generates spoken output and, demonstrating Kenenhrur's unique linguistic patterns.

```
// Define a phrase to be spoken, following Kenenhrur's distinct linguistic patterns
let spoken_greeting = Speech.Phrase([
    Speech.Word("Kfndfififirf", { emphasis: true }),
    Speech.Pause("short"),
    Speech.Word("jdyu"),
    Speech.Word("jhdhdjfo"),
    Speech.Word("jdyud"),
    Speech.Pause("medium"),
    Speech.Word("Djdydnfkifnfhcfghd"),
    Speech.Pause("short"),
    Speech.Word("N"),
    Speech.Word("dhgdid")
])


// Define a simple visual scene
```

```
let visual_greeting = Scene.create([
    Shape.text("Hello, Kenenhrur!", 0, 0, { font_size: 100, color: Color.blue() }),
    Shape.circle(50, 50, 20, { fill: Color.red() })
])

// Vocalize the greeting
Kenenhrur.Speech.Synth.vocalize(spoken_greeting)

// Render the visual scene to an image file
Kenenhrur.Image.Gen.render(visual_greeting, "hello_kenenhrur.png")

// Output confirmation
IO.print("Kenenhrur greeting generated!")
```

To run this code:

1. Save it as `hello_kenenhrur.ken`.
2. Open your Kenenhrur environment (terminal/IDE).
3. Execute: `kenenhrur run hello_kenenhrur.ken`

2. Kenenhrur Language Fundamentals

Kenenhrur is a multi-paradigm language, supporting functional, symbolic, and rule-based programming.

2.1. Basic Syntax

- Variables: Declared with `let`.
- Functions: Defined using `fn`.
- Data Structures: Kenenhrur supports immutable lists, maps, and symbolic expressions.

Example:

```
// Define a function that adds two numbers

let add_numbers = fn(a, b) -> a + b


// Create a list

let my_list = [1, 2, 3, "symbol"]


// Create a map

let my_map = { "key1": "value1", "key2": 123 }
```

2.2. Symbolic Expressions

Expressions are core to Kenenhrur's symbolic computation. They can represent data, code, or abstract concepts.

Example:

```
// A symbolic expression representing a mathematical formula

let formula = add(multiply(x, 2), 5)


// A symbolic expression representing a logical statement

let logical_statement = and(is_human(Alice), can_breathe(Alice))
```

3. Core Libraries

Kenenhrur's power lies in its specialized core libraries:

3.1. `Kenenhrur.Symbolic` - Symbolic Computation & Data Structures

This library provides the tools for pattern matching, transformation, and manipulation of symbolic data.

- `Kenenhrur.Symbolic.match(pattern, data)`:
    - Purpose: Matches a pattern against data, returning bindings.
    - Example:

```
let pattern_point = Point(X, Y)

let data_point = Point(10, 20)

Kenenhrur.Symbolic.match(pattern_point, data_point)

// -> { "X": 10, "Y": 20 }
```

    -
- `Kenenhrur.Symbolic.transform(expression, rule_set)`:
    - Purpose: Applies transformation rules to an expression.
    - Example:

```
let rules_simplify = Rule.Set([

  Rule.define("add_zero", IF: add(X, 0), THEN: X),

  Rule.define("mult_one", IF: multiply(X, 1), THEN: X)

])

Kenenhrur.Symbolic.transform(add(multiply(A, 1), 0), rules_simplify)

// -> A
```

    -

3.2. `Kenenhrur.Rules` - Rule-Based Programming

Enables declarative reasoning and inference.

- `Kenenhrur.Rules.define_rule(name, conditions, actions)`:
    - Purpose: Defines a new rule in the inference engine.
    - Example:

```
Kenenhrur.Rules.define_rule("child_can_play",
  IF: age(X, A) and less_than(A, 10),
  THEN: assert(can(X, "play"))
)
```

- `Kenenhrur.Rules.assert(fact)`:
  - Purpose: Adds a fact to the knowledge base.
  - Example:

```
Kenenhrur.Rules.assert(age(Sarah, 7))
// -> Might trigger "child_can_play" to assert can(Sarah, "play")
```

3.3. `Kenenhrur.Speech.Synth` - Expressive Voice Generation

Dedicated to high-quality, natural-sounding computer-generated speech, reflecting Kenenhrur's unique linguistic patterns.

- `Kenenhrur.Speech.Synth.vocalize(expression)`:
  - Purpose: Speaks a Kenenhrur expression, respecting phonetic and prosodic patterns.
  - Example:

```
let specific_phrase = Speech.Phrase([
  Speech.Word("Kfndfifirf"),
  Speech.Word("jdyu"),
  Speech.Word("jhdhdjfo"),
  Speech.Word("jdyud")
])
```

`Kenenhrur.Speech.Synth.vocalize(specific_phrase)`

    -

- `Kenenhrur.Speech.Synth.speak(text, options)`:
    - Purpose: Speaks plain text with customizable options.
    - Options: `rate`, `pitch`, `volume`, `emotional_tone`.
    - Example:

```
Kenenhrur.Speech.Synth.speak("Djdydnfkifnfhcfghd. N dhgdid.", { rate: 0.9, pitch: 1.05 })
```

    -

3.4. `Kenenhrur.Image.Gen` - Generative Visual Creation

Tools for procedural image generation and manipulation.

- `Kenenhrur.Image.Gen.render(scene_graph, filename)`:
    - Purpose: Renders a symbolic scene description to an image file.
    - Example:

```
let my_complex_scene = Scene.create([
  Shape.fractal("mandelbrot", { center_x: -0.5, zoom: 2 }),
  Shape.text("Generated by Kenenhrur", 10, 10, { color: Color.white() })
])
Kenenhrur.Image.Gen.render(my_complex_scene, "complex_fractal.png")
```

    -

- `Kenenhrur.Image.Gen.shape_from_rule(rules, options)`:
    - Purpose: Generates geometric patterns based on rule sets (e.g., L-systems).
    - Example:

```
let fern_rules = Rules.LSystem({ Axiom: "X", Rules: { "X": "F+[[X]-X]-F[-FX]+X", "F": "FF" } })
```

```
Kenenhrur.Image.Gen.shape_from_rule(fern_rules, { iterations: 5, angle: 25.7 })

// -> Generates a fern-like structure
```

3.5. `Kenenhrur.IO` - Input/Output & System Interaction

Handles file operations and external communication.

- `Kenenhrur.IO.read_file(path)`:
  - Purpose: Reads content from a file.
  - Example:

```
let data_from_file = Kenenhrur.IO.read_file("config.json")
```

- `Kenenhrur.IO.write_audio(audio_data, path, format)`:
  - Purpose: Writes generated audio data to a file.
  - Example:

```
Kenenhrur.IO.write_audio(my_generated_speech, "output_audio.wav", "WAV")
```

4. Advanced Topics & Community

- Extending Kenenhrur: Learn how to create your own libraries and integrate external services.
- Performance Optimization: Tips for writing efficient Kenenhrur programs.
- Community & Support: Coming soon

This documentation is continuously updated.