AI model interface documentation for AI Animation Bridge addon

The interface is named GeneralInterface and it is located in *interface/general_interface.py*. In order to assure compatibility of an AI model with our plugin, an implementation of this interface has to be created. For example implementations, please see the *models* folder.

Below you will find all methods of the interface with detailed explanations and examples.

1 Interface methods

1.1 check_frame_range

Method that is executed before generating the model results. If it returns false, no results will be generated, and the supplied error message will be displayed to the user.

It is not necessary to check whether start_frame and end_frame are in the right order and whether they fit within the scene frames. That is validated by our plugin. This method is supposed to facilitate limitations of your model, for example if it needs X frames of context before the start_frame, you should check for that. Otherwise, just always return true.

Input

- start_frame: int
  Start frame number selected by the user.
- end_frame: int
  End frame number selected by the user.
- scene_start_frame: int
  First frame number in the scene.
- scene_end_frame: int
  Last frame number in the scene.

Output

- tuple[bool, str]
  Bool representing whether the validation was successful and error message if applicable. Otherwise, return empty string.

Example implementation

```
def check_frame_range(self, start_frame, end_frame, scene_start_frame, scene_end_frame) -> tuple[bool, str]:
        if start_frame < scene_start_frame + 10:
            return (False, "Must be at least 10 frames before selected range.")
        if end_frame + 2 > scene_end_frame:
            return (False, "Must be at least 2 frames after selected range.")
        return (True, "")
```

1.2 is_skeleton_supported

Method that is executed before generating the model results. It is given a skeleton and should check whether it is supported by the model. If it returns false, no results will be generated.

In implementation, you can do simple checks, like in the example checking for one skeleton, or you can do more complicated checks if your model supports more than one skeleton.

Input

- skeleton: list[tuple[str, str]]
  Skeleton data. Each tuple in the list represents a bone. The first string is bone's name and the second is its parent's name. For root bones, parent name is None.

Output

- bool
  Boolean representing whether the skeleton is supported by the model.

Example implementation

```
def is_skeleton_supported(self, skeleton) -> bool:
        return sorted(skeleton) == sorted([
                ('Hips', None),
                ('LeftUpLeg', 'Hips'),
                ('LeftLeg', 'LeftUpLeg'),
                ('LeftFoot', 'LeftLeg'),
                ('LeftToe', 'LeftFoot'),
                ...
            ])
```

1.3 get_additional_infer_params

It returns a list of what optional parameters you want to be configurable in the UI when using your model. An empty list can be returned if there are none. These parameters are later passed to the infer_anim function as kwargs.

Supported variable types are basic types (bool, int, float, str) and torch.device.

Output

- list[tuple[type, str, str]]
  Each tuple represents one parameter. Type is the variable type of the parameter, the first string is the displayed name, and the second one is the description.

Example implementation

```
def get_additional_infer_params(self) -> list[tuple[type, str, str]]:
    return [
             (torch.device, "Device", "Select device to compute on"),
             (bool, "Post-processing", "Apply post-processing")
        ]
```

1.4 infer_anim

Most important method in the interface. Receives current animation data and returns inferred animation data for the given frames. You are supposed to return data for as many frames as there are between the start frame and end frame selected by the user.

Input

- anim_data: dict
  Dictionary with animation data of the skeleton. Structure including examples as described below.

Dictionary fields

- names: numpy array [str]
  List of names of each bone.

```
'names': array(['Hips', 'LeftUpLeg', 'LeftLeg', 'LeftFoot', ...])
```

- parents: numpy array [int]
  Index of parent bone for each of the bones in the names array. When bone has no parent it is -1.

```
'parents': array([-1,  0,  1,  2,  3,  0,  5,  6,  7,  0,  9, 10, 11, ...])
```

- offsets: 2D numpy array [float]
  Position of each bone at frame zero.

```
'offsets': array([
      [-2.22377106e+02, -4.44538918e+01,  8.58522263e+01], # Bone 1
      [ 1.03454590e-01, -1.05485001e+01,  1.85783386e+00], # Bone 2
      [ 4.35000000e+01,  7.62939453e-06, -2.28881836e-05], # Bone 3
      ...                                                  # etc.
    ])
```

- positions: 3D numpy array [float]
  Positions for each frame for each bone in the scene. The outer array is for frames and the more inner ones are for bones.

```
'positions': array([[
      [-2.22377106e+02,  8.58521957e+01,  4.44538918e+01],
      [ 1.03456497e-01,  1.85786176e+00,  1.05485029e+01],
      [ 4.35000153e+01, -2.09808350e-05, -1.16825104e-05],
      ...
    ]])
```

- rotations: 4D numpy array [float]
  Rotations for each frame for each bone in the scene. The outer array is for frames and the more inner ones are for bones. Rotations are represented as 3x3 rotation matrices.

```
'rotations': array([[
      [
        [ 0.04132594,  0.91554169,  0.40009447],
        [ 0.98721817, -0.09910521,  0.12481361],
        [ 0.15392351,  0.38982249, -0.90793501]
      ],
      [
        [-0.81039062,  0.57895811, -0.08985851],
        [ 0.58508468,  0.80774045, -0.07232762],
        [ 0.03070769, -0.11118846, -0.9933248 ]
      ],
      ...
    ]])
```

- start_frame: int
  Start frame number selected by the user.

- end_frame: int
  End frame number selected by the user.

- kwargs
  Additional parameters as defined by get_additional_infer_params method.

Output

- tuple[list, list]
  Tuple of inferred positions and inferred rotations by your model. Structure as described below.

  Tuple fields

  – inferred_positions: 3D numpy array [float]
  Output positions inferred by your model to be applied to the object. Positions are for each frame for each bone in the scene. The outer array is for frames and the more inner ones are for bones.

```
inferred_positions = array([
    [
        [-2.27917297e+02,  8.50552673e+01,  4.40606461e+01],
        [ 1.03448153e-01,  1.85784650e+00,  1.05485020e+01],
        ...
    ],
    [
        [-2.30289497e+02,  8.45934296e+01,  4.36578217e+01],
        [1.03439420e-01,   1.85783875e+00,  1.05485072e+01],
         ...
    ],
  ...
])
```

  – inferred_rotations: 3D numpy array [float]
  Output rotations inferred by your model to be applied to the object. Rotations are for each frame for each bone in the scene. The outer array is for frames and the more inner ones are for bones. The rotations should be of Euler type, with ZYX order in degrees.

```
inferred_rotations = array([
    [
        [ 1.62996522e+02,  1.03901119e+01,  9.39967920e+01],
        [-1.77134607e+02,  4.32209362e-01,  3.87051794e+01],
        ...
    ],
    [
        [ 1.64700164e+02,  1.06121627e+01,  9.44343803e+01],
        [-1.79701658e+02,  1.17249186e+00,  3.89786331e+01],
         ...
    ],
  ...
```

```
])
```

Example implementation

```python
def infer_anim(self, anim_data, start_frame, end_frame, **kwargs):
        # Example kwargs retrieval with default values
        device = kwargs.get("Device", "cpu")
        post_processing = kwargs.get("Post-processing", False)

        # Animation data retrieval
        positions = anim_data["positions"][start_frame:]
        rotations = anim_data["rotations"][start_frame:]
        offsets = anim_data["offsets"])
        parents = anim_data["parents"])

        # Your model evaluation
        inferred_pos, inferred_rot = your_model.evaluate(offsets, parents,
    positions, rotations, device, post_processing)

        return inferred_pos, inferred_rot
```