Metadata-Based Rigging Pipeline

A Thesis
by
Dmitrii Shevchenko

MSc Computer Animation and Visual Effects
N.C.C.A. - Bournemouth University
August 2021

Table of Contents

List of figures

1 Introduction

Rigging is a vital step in any 3D content production pipeline of visual effects and game studios. It connects modelling and animation stages, providing animators with a rig – usually a set of controllers driving the deformation of the asset's model. The manual creation of rig is complex and includes great number of repetitive tasks.

In order to increase riggers' productivity, procedural solutions are implemented for automation of rig building and improved iteration between rigging, modelling and animation departments. These solutions range from simple auto rig scripts to proper software that handle rig creation from start to end. However, many of these systems do not provide rigger with indestructible workflow, in which rig can be revised while already being used in production. Furthermore, often there is no encoding of the rig data besides in scene description, making version control systems not applicable.

This paper presents a modular rigging pipeline that enables user to create rigs using either Python API directly or via node-based interface. The system aims to be extensible, allowing rigger to add new custom rig components or extend already existing ones. In order to be tool-oriented this pipeline utilizes metadata concept and component to Python class bindings which leads to consistent functionality for any additional tools.

This system is available as a plug-in for the Autodesk Maya and verified as stable on Maya versions 2019.3 and 2020.4. It utilizes PyMEL Python library in order manage asset files and create metadata-based rig components. Python and PySide2 are used for creation of node-based rig editor and additional supporting tools. JSON is used for rig editor scene serialization.

Developed pipeline provides rigger with a tool set for creating and editing rigs in indestructible manner, that is uncommon outside of in-house proprietary plugins. Rig builder interface is simple yet flexible enough to produce a large range of rig types and if user decides to customize low level rig behaviour further there is a Python API available.

2 Related work

2.1 Modular rigging

In the recent years there has been a rise in modular rigging systems being developed as in house tools within Visual Effects and Games studios and free open-source projects available for general public. This could be seen as response to a growing complexity of digital assets that require animation as well as general technology advancement allowing for increased fidelity of such assets.

Some of the earliest examples of such systems that were publicized are "Block Party" developed by "Industrial Light & Magic's" and procedural rigging system by "Blue Sky Studios". Both introduced rigging concepts that were later reimagined and polished by other studios and independent developers. One such key concept is splitting rig into independent components that encapsulate specific rig logic, can be created independently from other components and are reusable for the different regions of the character.

As for more recent examples, plugins that are currently considered to be sufficient for most rigging needs are Advanced Skeleton and mGear (Figure 1). Both extend on the earlier defined principle of rig modularity. Advanced Skeleton plugin was developed by "Animation Studios" and has it price starting at $750 USD. It provides user with a single dialog interface that manages the whole rig creation process, including definition of the "fit skeleton", individual modules creation and post build task like deformer weights import. By contrast, mGear is open-source project, uses MIT license and positions itself as a "rigging framework". Being a framework mGear consists of a wide range of rigging tools that build on its core functionality as well as several tools for animators to improve their interaction with produced rig.

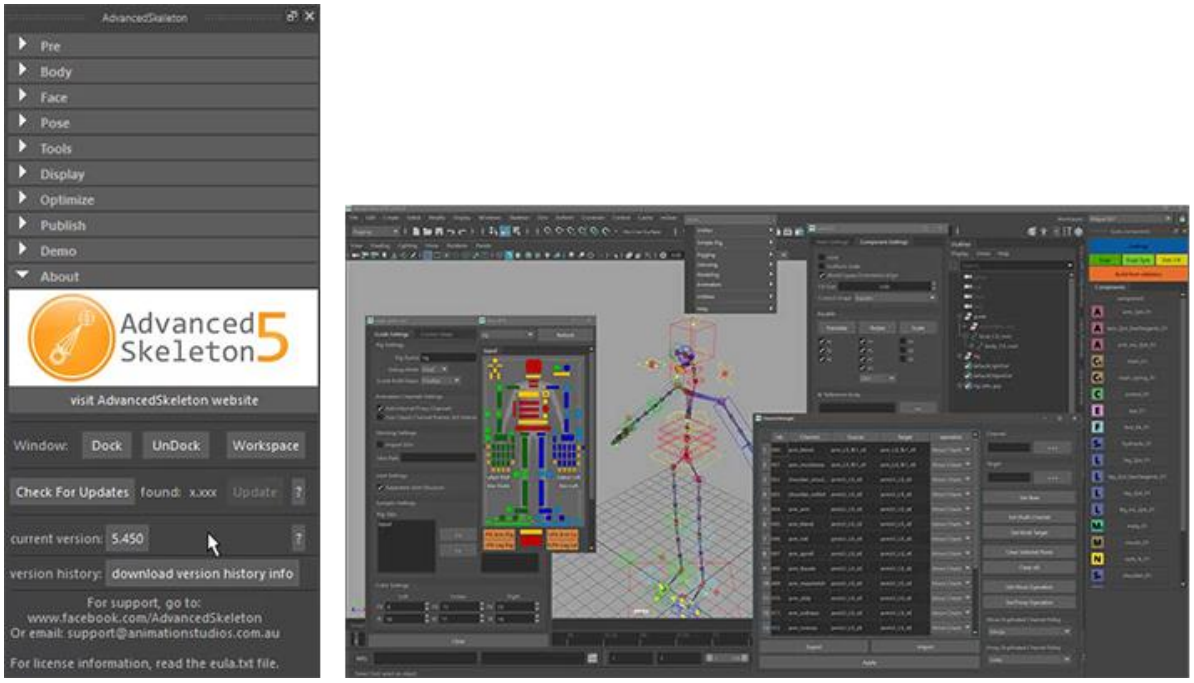


Figure 1: Advanced Skeleton and mGear plugins, (Animation Studios, n.d.) (Campos, et al., 2021).

2.2 Metadata

An important underlying concept for the next rigging system is metadata. According to (TechTerms, 2021) the general definition this term would be: “Metadata describes other data. It provides information about a certain item's content”. Applying this definition to modular rigging would result in metadata holding description of rigging component allowing for ease of lookup and relationship connections between components.

Early example of a rigging system that utilizes metadata concept is rigging pipeline presented by David Hunt at GDC 2009 (Hunt, 2009), who at time was working at video game company called Bungie. This system was implemented using Python inside Autodesk Maya and was relying on traversal of internal rig network by creating additional MetaNodes inside Maya’s Dependency Graph allowing for markup and connection of rig components.

In the later presentation at GDC 2015 David Hunt together with Forrest Sderlind presented a significantly improved version of this pipeline (Hunt & Forrest, 2015). In addition to features of the previous iteration the new version allowed for a linking between rig component inside Maya’s scene and corresponding Python object class that was used to generate set component. Coupling was achieved by storing class name as string attribute of MetaNode inside Maya’s scene. This allowed system to instance a Python object that would represent a rig component and have ability to retrieve and modify important Maya’s scene objects by calling object instance’s methods. Furthermore, it enabled easy component lookup by Python type including subclassed types.

2.3 Node-based interfaces

These days most 3D content creation packages provide user with node-based system, which could be optional or integral part of the application. Examples of such systems are Maya’s dependency graph, Houdini’s procedural graph and Unreal Engine’s blueprints. These graphs can be divided into two categories: evaluation and execution.

In the evaluation graph every node is computed and contributes to the result. Such graphs usually implement node “dirty” state for faster evaluation by ignoring clean nodes that have not had any attribute changes since last evaluation. This means that any node with an invalid data would result in graph evaluation error. Applications using this type of graph include Maya, Houdini, Substance Designer, Nuke (Figure 2).

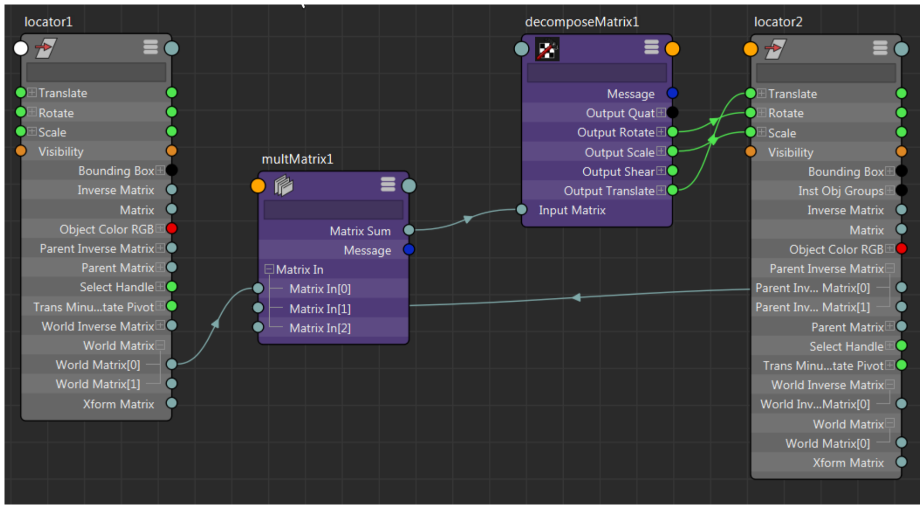


Figure 2: Autodesk Maya node editor (evaluation graph) (Shotarov, 2017).

On the other hand, execution graph has a concept of the execution chain that defines the order in which nodes will be computed. In most cases this chain is constructed by connecting 'execute' attributes of multiple nodes. As a result, any node that is not a part of this chain will be completely ignored even if it has invalid data at graph execution time.
This type of graphs is common among applications that implement visual scripting such as Unreal and Unity game engines (Figure 3).

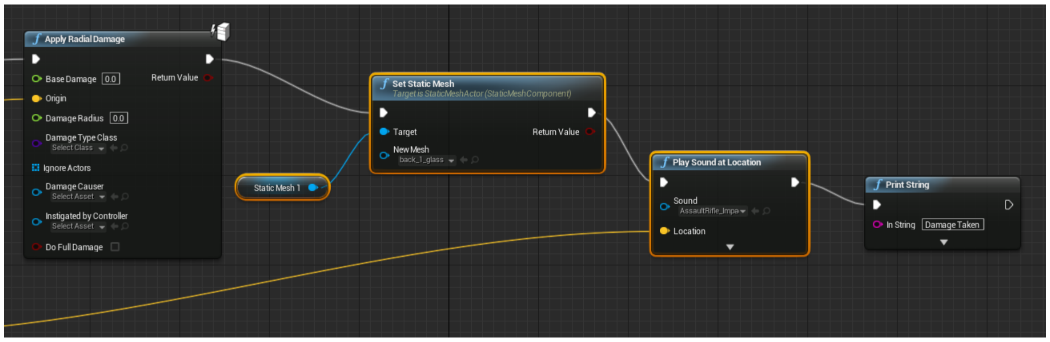


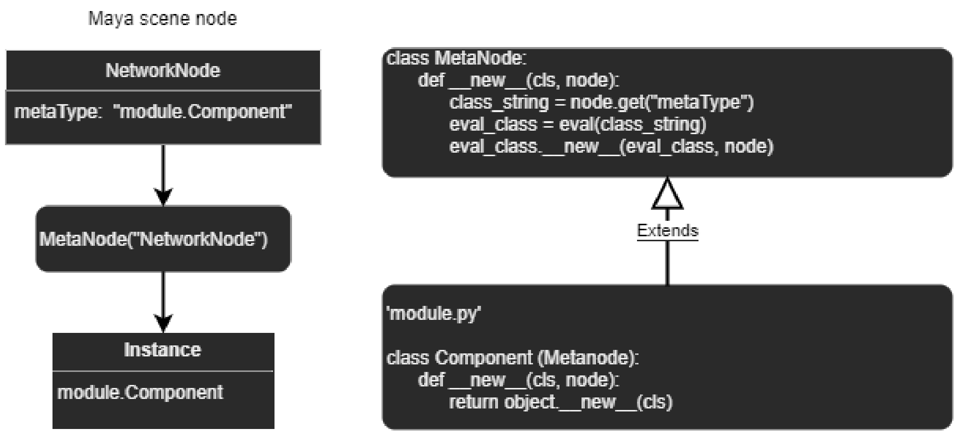
Figure 3: Unreal Engine 4 Blueprints editor (execution graph) (Epic Games, n.d.).

3 Technical background

3.1 Python __new__ method

According to definition from (TechTerms, 2021): "Whenever a class is instantiated __new__ and __init__ methods are called. __new__ method will be called when an object is created and __init__ method will be called to initialize the object." Modifying this core method essentially allows to control which class instance will be created as a result of class constructor call.

One of the key ideas, introduces in previously mentioned Bungie's 2015 rigging pipeline is modifying base MetaNode class's __new__ method to allow instancing of correct rigging component class from stored metadata. It is achieved by first writing component's class module and name as string to the attribute on component's network node. Later using this string data, a reference to correct class can be evaluated using Python's eval function and used to call correct __new__ method (Figure 4). It is also important to override __new__ method back to original for any subclass in order to avoid infinite recursion.



```
class MetaNode:
    def __new__(cls, node):
        class_string = node.get("metaType")
        eval_class = eval(class_string)
        eval_class.__new__(eval_class, node)
```

```
'module.py'

class Component (Metanode):
    def __new__(cls, node):
        return object.__new__(cls)
```

Figure 4: __new__ method overriding for subclass instancing.

3.2 Creating advanced Maya based interfaces

It is possible to point out three main APIs that allow Maya's functionality to be extended: MEL (Maya Embedded Language), Python and C++ (Figure 5). MEL can be used for creating basic interfaces but is not suitable for large code bases and plugin development for Maya, as it lacks many features of modern programming languages and does not have access to internal Maya API. C++ would be a great choice for plugin development but is an overkill in case if plugin relies on commands module instead of internal API. Python on the other hand has direct access to both: commands and internal API, making it the best choice for plugins that do not require computation speed of C++.

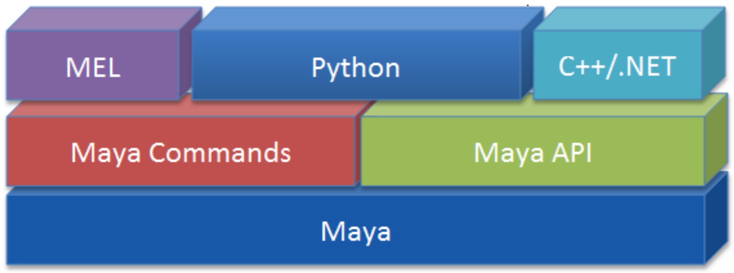


Figure 5: Available Autodesk Maya APIs (Autodesk, 2017).

When it comes to creating Maya based user interfaces with Python, user is presented with commands module's UI functions that can be enough for basic interface prototyping using prebuilt widgets, but do not allow for creation of custom widgets or behaviour modification of existing ones. Another Python library that can be used and comes with Maya is PySide2. It is an official Python binding of Qt and is far more flexible than commands module in terms of interface creation, as it allows developer to create custom widgets that might include advanced features like graphics drawing or docking functionalities.

4 Implementation

4.1 Overview

The rigging pipeline described in this thesis was named "Luna" and consists of four different submodules in order to provide all its functionality (Figure 6). The first module ("luna") includes modules for rigging workspace management, unit testing, custom maya menus and minor interfaces, static definitions, additional tools and common Python utility functions. The second module ("luna_rig") is responsible for handling rig creation and import / export operations of rig related data. The third module ("luna_configer") provides user with graphical interface for managing various settings of the pipeline. The fourth and final module ("luna_builder") is a node-based interface for rig creation.

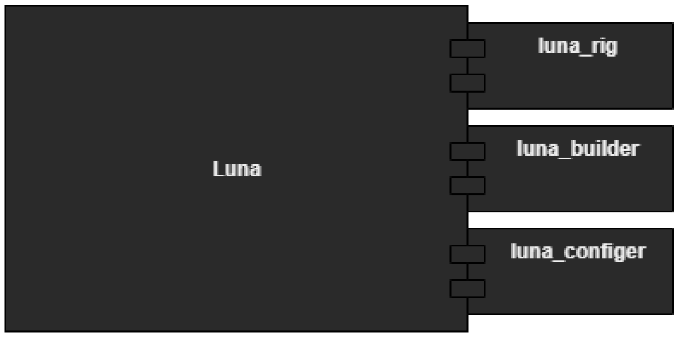


Figure 6: System modules.

4.2 Logging and configuration

In order to improve developer's experience and future system flexibility the first modules to implement were Logger and Config. Logger is a singleton class that wraps the functionality of Python built-in logging library. This class mainly consists of class methods that log messages at different severity levels. By default, Luna additionally logs errors and exceptions to rotating log file "luna.log" located at the root of the plugin directory.

Config class consists of class methods that manage configuration of the pipeline stored in a json file. A decision was use external json instead of Maya's built-in "optionVars" to avoid Maya's preference file corruption and make it more readable in general. There are two configuration files Luna is keeping track of – a default configuration that comes with download and user configuration that is generated the first time plugin is loaded. In case if user configuration is deleted it will be regenerated from default one on plugin load or if there was an attempt to access any of configuration values. In order to avoid file IO performance issues this class also stores cached configuration dictionary for faster access.

4.3 Plugin menu

Most plugin functionality is available through menu located at Maya's main menu bar after plugin is successfully loaded (Figure 7). From this menu user can access core tool dialogs like builder or configuration, supporting and registered external tools, and finally developer specific functions like unit tests execution.

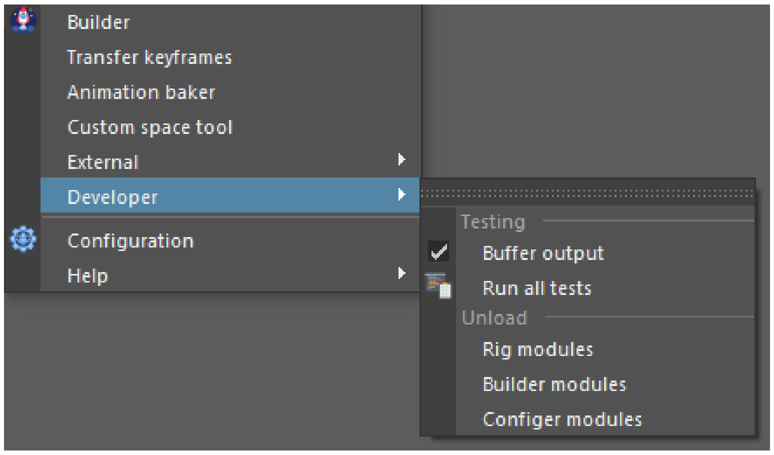


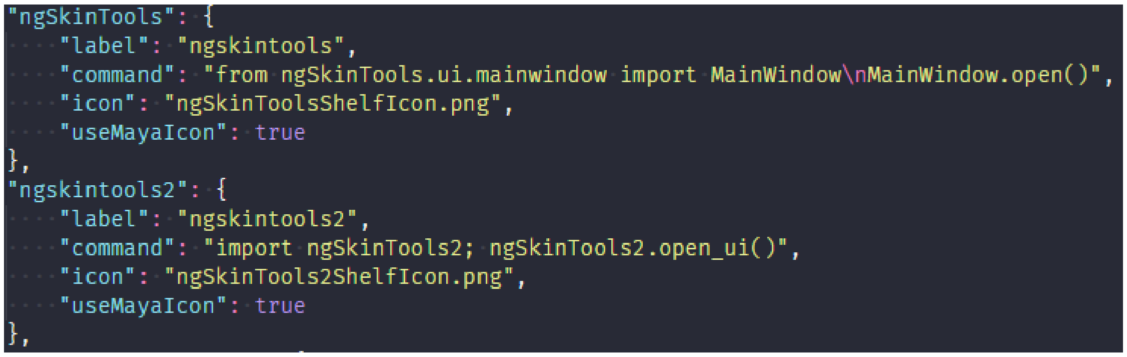
Figure 7: Plugin main menu.

4.4 External tools registration

The pipeline menu supports registration of external tools by adding their description into a json file that serves as a registry (Figure 7). If tool is found during plugin initialization process it will be available through a "External" submenu of main Luna menu (Figure 8). In order to be eligible for registration it is required for tool to be implemented as Maya module due to automatic module lookup that happens when menu is populated. As registration is per module, it is only possible to register a single tool for a module, this is because of the key-value link between module name in Maya's modules register and command string in external tools register. So, it is only suitable for utility tools or other plugins shortcuts.

By default, the following tools are present in the register:

- ngSkinTools
- ngSkinTools2
- dsRenamingTool
- sl_history
- texture_retargeter
- ds_playblast



```
"ngSkinTools": {
    "label": "ngskintools",
    "command": "from ngSkinTools.ui.mainwindow import MainWindow\nMainWindow.open()",
    "icon": "ngSkinToolsShelfIcon.png",
    "useMayaIcon": true
},
"ngskintools2": {
    "label": "ngskintools2",
    "command": "import ngSkinTools2; ngSkinTools2.open_ui()",
    "icon": "ngSkinTools2ShelfIcon.png",
    "useMayaIcon": true
},
```

Figure 8: External tools registration of ngSkinTools plugin

4.5 Custom dynamic marking menu

At plugin load a new marking menu is registered for Ctrl+Alt+MMB shortcut. This menu has two operation modes: animator and rigger. The mode can be switched through configuration dialog (Figure 14) or directly in config file. The population of the menu is done dynamically based on scene selection and operation mode by adding appropriate action groups (Figure 10).

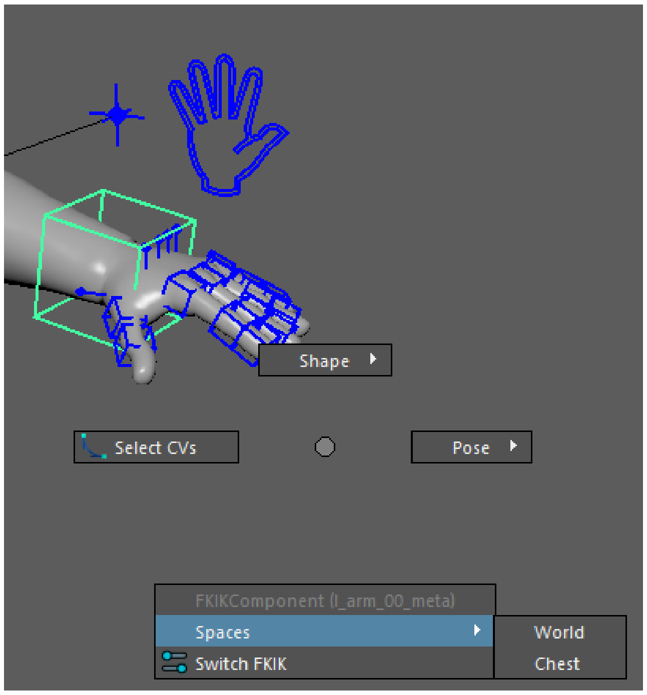


Figure 9: Dynamic marking menu (FKIKComponent control selected).

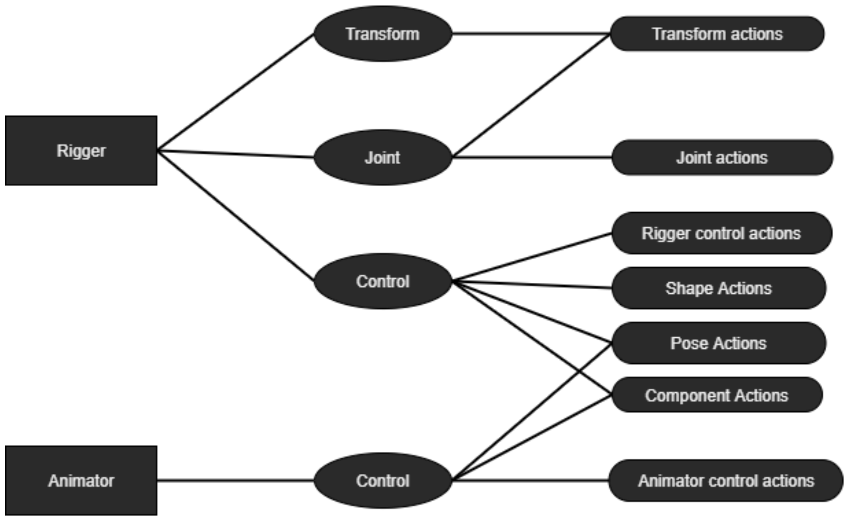


Figure 10: Marking menu action groups diagram.

The breakdown of action groups is as follows:

- Pose actions:
  - Mirror pose (Behaviour) – finds matching control on the other side and copies selected control's pose and mirrors other control transform matrix.
  - Mirror pose (No Behaviour) - Identical to "Mirror pose (Behaviour)" but does not mirror transform matrix.
  - Asset bind pose – reverts all character's controls to their recorded bind pose.
  - Component bind pose – reverts other controls in the component to their bind pose.
  - Control bind pose – reverts selected control to its bind pose.
  - Component actions:
  - Populated dynamically by calling component's "actions_dict" property.
  - If control has attribute "space" - space sub menu will be created with list of available spaces to switch to. Switch is done with matching, so control will stay in place.
- Shape actions:
  - Load shape – opens saved shapes directory where user can choose a new shape for selected controls.
  - Copy shape – copies selected control's shape to a temporary file.
  - Paste shape – assigns shape from copied shape file if it exists.
  - Copy colour - copies selected control's colour to a temporary file.
  - Paste colour – assigns shape colour from copied colour file if it exists.
  - Mirror shape YZ – mirrors selected shape over local YZ plane.
  - Flip shape YZ – flips shape over local YZ plane.
  - Mirror shape to opposite control - finds control on the opposite side, copies shape to it and then mirrors it over local YZ plane.
- Transform actions:
  - Create locator – creates a locator with matching position and rotation
  - Match position – matches position of last selection to first selection
  - Match rotation – matches rotation of last selection to first selection
  - Match object centre – snaps all selected objects to first object's centre
  - Match components centre – matches last selected object's transform to centroid of all selected components.
- Joint actions
  - Joint chain from selection – creates joints hierarchy from selection in order of selection.
  - Mirror joints – mirror joints and automatically rename to opposite side.
  - Sub menu for "Transform actions"
- Rigger control actions
  - Select CVs – select all control points of last selected control shape.
  - Sub menu for "Shape actions"
  - Sub menu for "Pose actions"
  - Sub menu for "Component actions"
- Animator control actions
  - Select component control – selects all controls of component that current selected control is member of.
  - Key component controls – adds keyframe for all controls of component that current selected control is member of.

- Sub menu for “Pose actions”
- Sub menu for “Component actions”

4.6 Rigging workspace

In order to encapsulate rigging process out of main production pipeline the system organises rigs by projects and assets.

4.6.1Project

Rigging project is represented by Python class that manages a directory on user’s machine. An empty project will contain only two files: luna.proj tag and project metadata. Tag file is used to identify a valid project directory and metadata holds a categorized list of assets and creation time stamp.

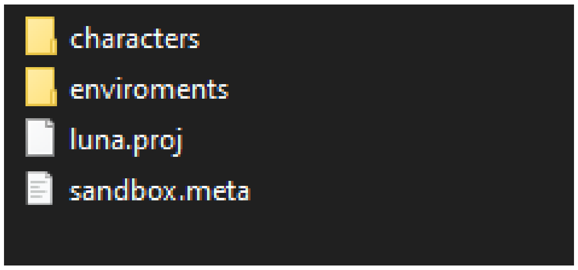


Figure 11: Rigging project directory example.

4.6.2Asset

Rigging asset also has a Python class associated with it. When asset is set it will create a category directory for itself (e.g. characters) if one does not already exist in the project, a subdirectory with asset name and any of the following directories and files that are missing:

- Build – rig builder saved rig files.
- Controls – exported control shapes
- Data
    - Blendshapes – exported blendshape files and mapping json file.
    - Driven poses – exported set driven key poses.
    - Mocap – exported HIK definitions.
    - Psd – exported pose space deformers.
    - Sdk_correctives – exported set driven key corrective poses
    - Xgen – exported xgen data. (Not implemented)
- Rig – Saved rig versions
- Settings – exported node presets. (Not implemented)
- Skeleton – asset skeleton and guides scenes.
- Weights – exported weights for various deformers.
- AssetName.meta – metadata json file that includes asset’s name, type, model path, creation and last modified time stamps.

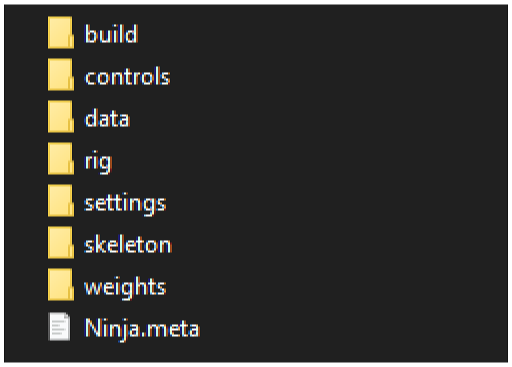


Figure 12: Rigging asset directory example.

4.6.3Heads up display

The change of current project or asset triggers an update of rigging heads-up display (Figure 13). It informs user of current project and asset names and by default takes section 7 block 5 of Maya’s heads up displays layout. In case if layout slot is already taken by another display user will be informed about it and given a suggestion to try changing section or block values via configuration dialog (Figure 14).

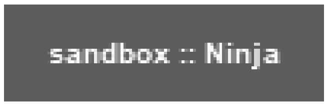


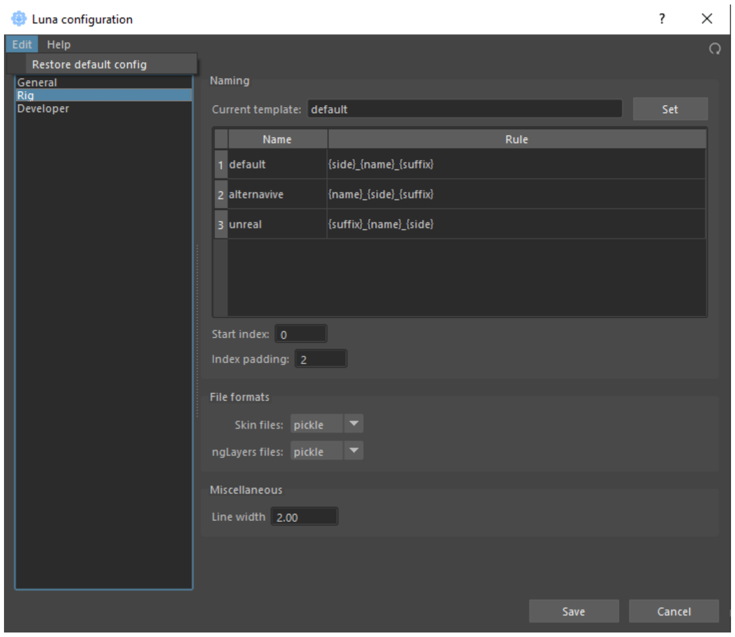
Figure 13: Rigging workspace heads up display (Project :: Asset).

4.7 Configuration dialog

An interface presented at Figure 14 allows user to manage pipeline configuration without need to adjust json file directly. Dialog is constructed of a QListWidget holding list of categories and a QStackedWidget that contains actual configuration pages. Changing selected index in categories list will set corresponding index in QStackedWidget. Each configuration page is a widget inheriting from base PageWidget class that requires methods "load_config" and "save_config" to be implemented.

Currently available settings:

- Logging level
- Unit testing parameters
- Python command port
- Rig naming template
- File format choice for skin and ngSkinTools data
- Controls line width
- Marking menu mode
- Heads up display position



Current template: default

	Name	Rule
1	default	{side}_{name}_{suffix}
2	alternavive	{name}_{side}_{suffix}
3	unreal	{suffix}_{name}_{side}

Start index: 0
Index padding: 2
Skin files: pickle
ngLayers files: pickle
Line width 2.00

Figure 14: Configuration dialog.

4.8 MetaNode rig architecture

Rig produced by this system is based on two main classes: MetaNode and Control. Both classes use class level method "create" for instance creation and constructor is used to initialize object from existing nodes in the scene.

When an MetaNode is instanced using "create" method it creates a Maya network node and adds the following attributes to it: metaType, metaChildren, metaParent and tag. MetaType attribute value is set to a string holding class's full name, making it usable with Python's eval function. Due to how class inheritance works any subclass will have its full name used for metaType value instead of default MetaNode class (Figure 16).

Class Component is a subclass of MetaNode and implements basic component functionality like storing and deleting utility nodes and settings. AnimComponent inherits from Component class and represents component that has controls and can be animated. Any other component like FKIK, FK, Spine etc. would inherit from AnimComponent and override create and attachment methods.

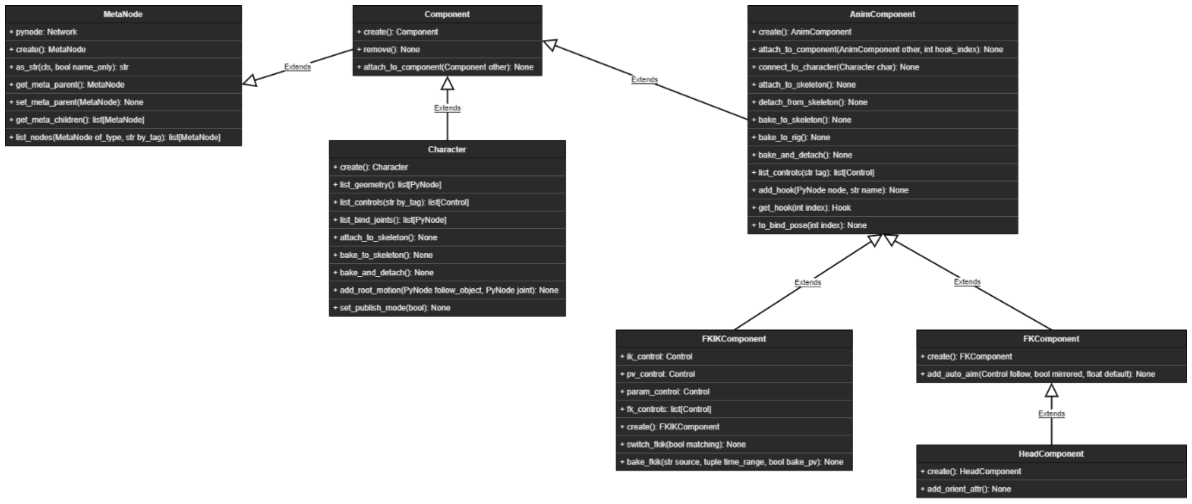
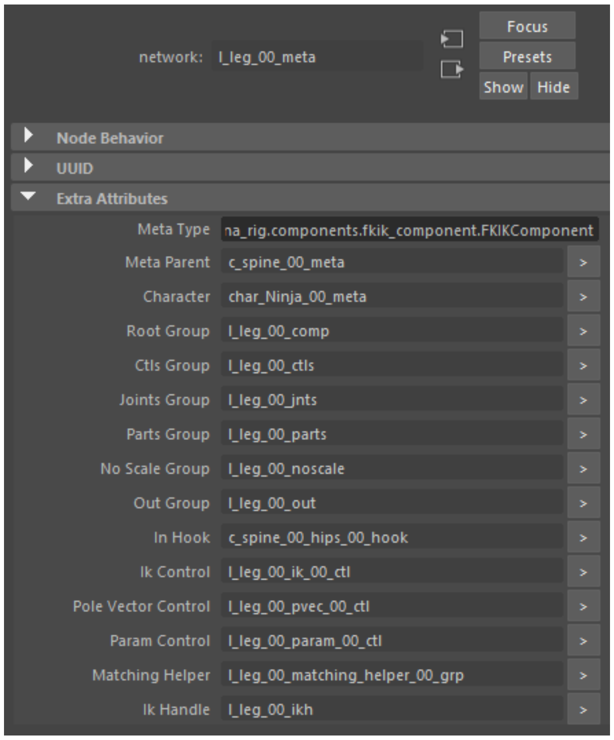

MetaNode
+ pynode: Network
+ create(): MetaNode
+ as_str(cls, bool name_only): str
+ get_meta_parent(): MetaNode
+ set_meta_parent(MetaNode): None
+ get_meta_children(): list[MetaNode]
+ list_nodes(MetaNode of_type, str by_tag): list[MetaNode]
Extends
Component
+ create(): Component
+ remove(): None
+ attach_to_component(Component other): None
Extends
Character
+ create(): Character
+ list_geometry(): list[PyNode]
+ list_controls(str by_tag): list[Control]
+ list_bind_joints(): list[PyNode]
+ attach_to_skeleton(): None
+ bake_to_skeleton(): None
+ bake_and_detach(): None
+ add_root_motion(PyNode follow_object, PyNode joint): None
+ set_publish_mode(bool): None
Extends
AnimComponent
+ create(): AnimComponent
+ attach_to_component(AnimComponent other, int hook_index): None
+ connect_to_character(Character char): None
+ attach_to_skeleton(): None
+ detach_from_skeleton(): None
+ bake_to_skeleton(): None
+ bake_to_rig(): None
+ bake_and_detach(): None
+ list_controls(str tag): list[Control]
+ add_hook(PyNode node, str name): None
+ get_hook(int index): Hook
+ to_bind_pose(int index): None
Extends
Extends
FKIKComponent
+ ik_control: Control
+ pv_control: Control
+ param_control: Control
+ fk_controls: list[Control]
+ create(): FKIKComponent
+ switch_fkik(bool matching): None
+ bake_fkik(str source, tuple time_range, bool bake_pv): None
FKComponent
+ create(): FKComponent
+ add_auto_aim(Control follow, bool mirrored, float default): None
Extends
HeadComponent
+ create(): HeadComponent
+ add_orient_attr(): None

Figure 15: Base inheritance diagram.

The attachment of components is performed using meta parent/child link and hook system. Using AnimComponent's "add_hook" method rigger can register joint or control as a hook which will result in Hook object being created and stored per component instance. Hook represents a transform node that is point and orient constrained to object that was passed to "add_hook" method. Hooks are stored on AnimComponent's network node using network node using message connections and can be retrieved using "get_hook(hook_index)" method.

Create method of most AnimComponents will have an optional "hook" argument that expects an integer value for hook index that will determine attach point. For example, Spine component registers hips joint as hook with index 1, so if during FKIK component creation Spine instance is passed as meta parent and hook argument was set to 1 - created FKIK instance will be following hips joint of the spine (Figure 17).



network: l_leg_00_meta

Meta Type	na_rig.components.fkik_component.FKIKComponent
Meta Parent	c_spine_00_meta
Character	char_Ninja_00_meta
Root Group	l_leg_00_comp
Ctls Group	l_leg_00_ctls
Joints Group	l_leg_00_jnts
Parts Group	l_leg_00_parts
No Scale Group	l_leg_00_noscale
Out Group	l_leg_00_out
In Hook	c_spine_00_hips_00_hook
Ik Control	l_leg_00_ik_00_ctl
Pole Vector Control	l_leg_00_pvec_00_ctl
Param Control	l_leg_00_param_00_ctl
Matching Helper	l_leg_00_matching_helper_00_grp
Ik Handle	l_leg_00_ikh

Figure 16: Created FKIKComponent attributes.

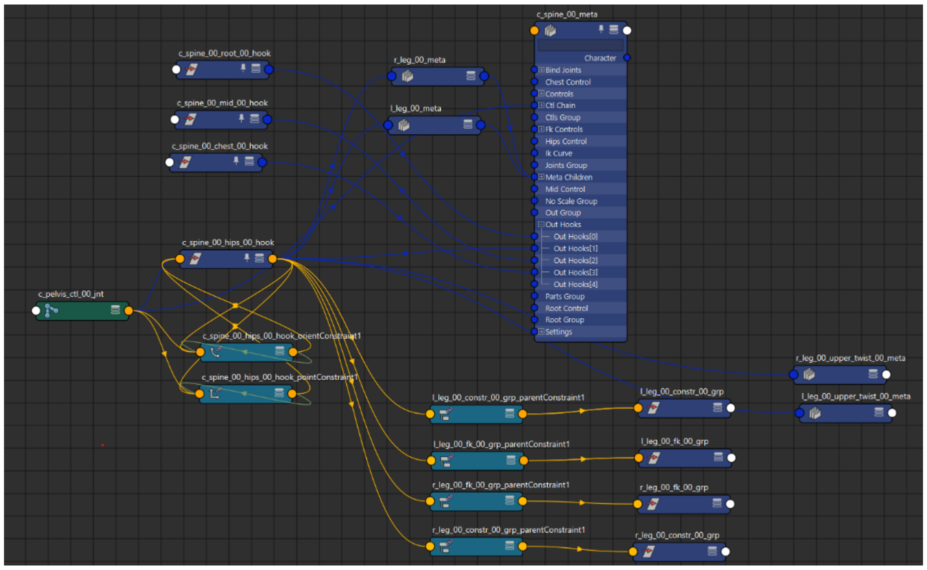


Figure 17: FKIKComponent attachment to SpineComponent Hips hook.

Character class inherits from Component as it requires a unique DAG hierarchy and is unique per rig (Figure 18). Its outliner hierarchy consists of top level "character_node" control, which has following child transforms:

- Control_rig – Parent for any AnimComponent root node.
- Deformation_rig – Parent for character's skeleton. Rigger is expected to parent appropriate joint hierarchy during build.
- Locators_grp – holds any global character locator.
- Util_grp – Parent for utility nodes.
- Geometry_grp – Parent for character's geometry.

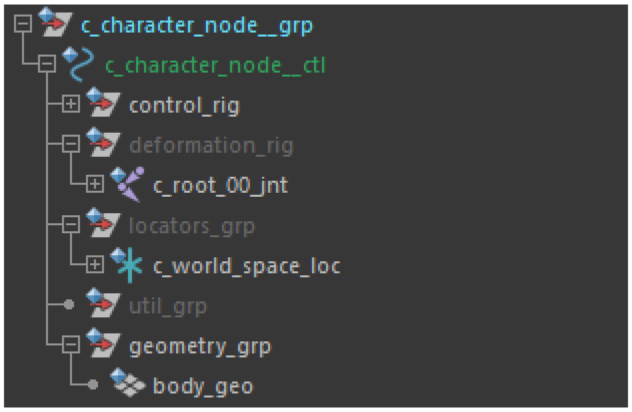


Figure 18: Character component outliner hierarchy.

There are a few reusable rigging components that come with this pipeline:

- FKComponent - suitable for any joint chain that requires forward kinematics setup.
- HeadComponent – subclass of FKComponent with several adjustment to make head creation more convenient.
- IKComponent - suitable for any joint chain that requires inverse kinematics setup.
- IKSplineComponent – spline-based IK component perfect for rigging tails, tentacles etc.
- FKIKComponent – component with an option to switch between forward and inverse kinematic modes. Supports FKIK matching and baking of animation between modes.
- FKDynamicsComponent – utilizes Maya's hair system to add a dynamics layer to FKComponent.
- FKIKSpineComponent – spine setup with layered FK and IK controls.
- FootComponent – reverse foot setup.
- TwistComponent – creates additional configurable number of twist joints that have twist distribution between start and end objects.
- HandComponent – base component that has an option to create child FKComponents that act as fingers.
- EyeComponent – component with FK and aim controls.

- SimpleComponent – blank component that has ability to add existing controls or create new ones. Very efficient when creating simple rigs.
- RibbonComponent – utilizes NURBS surface and Maya's hair to drive deformation of generated bind joints.
- CorrectiveComponent – component used to create joint based corrective controls that rely on defined driven key relations.
- WireComponent – creates wire deformer and controls for given geometry.
- IKSplineStretch – not animatable component that implements stretch for spline-based IK components (e.g. FKIKSpineComponent or IKSplineComponent)
- IKStretchComponent – not animatable component that adds stretch for IK components (e.g. FKIKComponent or IKComponent)

4.9 PyBuild

PyBuild is a Python class that streamlines rig building for riggers that decide to use Luna's Python API directly. This class handles verifying current rigging project and asset, importing model and skeleton scenes and executing post build tasks like view adjustment. In order to create a new build user is expected to inherit from PyBuild and override it's "run" and "post" (optional) methods and finally create an instance of newly declared class to initiate rig build process (Figure 20).

The required parameters asset type and name are used for asset and character instances creation. Created character instance reference is stored as instance field on PyBuild object and can later be passed to any component creation method.

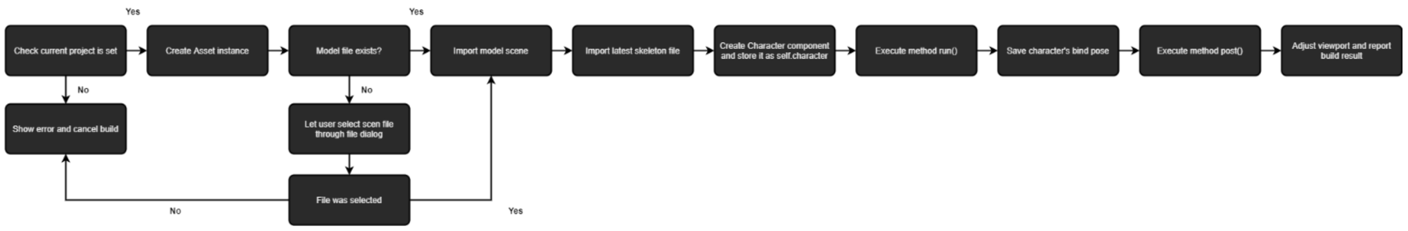
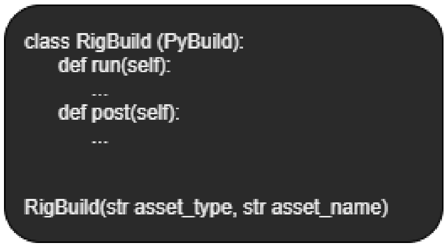

Check current project is set
Yes
Create Asset instance
Model file exists?
Yes
Import model scene
Import latest skeleton file
Create Character component and store it as self.character
Execute method run()
Save character's bind pose
Execute method post()
Adjust viewport and report build result
No
Show error and cancel build
No
Let user select scen file through file dialog
File was selected
No
Yes

Figure 19: PyBuild rig building process.



```
class RigBuild (PyBuild):
    def run(self):
        ...
    def post(self):
        ...


RigBuild(str asset_type, str asset_name)
```

Figure 20: PyBuild usage example.

4.10 Import / Export managers

During rig iteration process it is common for some rig data to be authored manually, this might include deformer weights, control shapes or node's attributes. To avoid problem of having to repeat any manual adjustment to the rig, user has an option to export current state of chosen data and import it back during next rig build iteration. This is done by utilizing data managing classes, that currently include:

- BlendshapeManager (shape data, does not include weights)
- CtlShapeManager (control NURBS curves)
- DrivenPoseManager (set driven key poses)
- NgLayersManager (skin weights layers data for ngSkinTools plugin)
- NgLayers2Manager (skin weights layers data for new ngSkinTools 2.0+)
- PsdManager (Maya's pose interpolators, will also export any connected blend shapes)
- SDKCorrectiveManager (poses of CorrectiveComponent)
- SkinManager (skin cluster weights)
- DeltaMushManager (delta mush deformer weights)

4.11 Rig builder

4.11.1 Overview

In case if user prefers a more visual approach to building a rig, there is an option to use node-based Luna builder editor (Figure 21). In terms of base functionality this editor takes inspiration from Unreal Engine blueprint editor that is a visual scripting tool. Like blueprint editor, Luna builder is an execution graph that allows user to script rig building process in a visual way. Each node in the scene represents either a component or a function. Connecting "EXEC" sockets of nodes defines the execution order of the graph. User can modify node's input data by selecting it and adjusting available values in attributes tab or connecting other node's outputs.

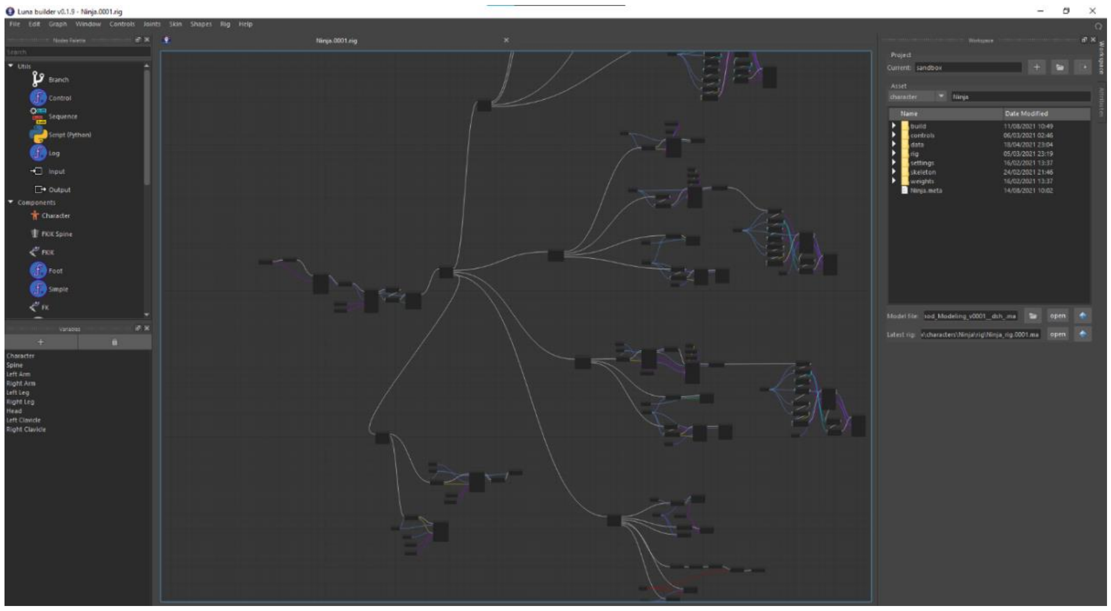


Figure 21: Luna rig builder interface.

4.11.2 Editor's plugin system

The node editor is designed to be extensible by additional plugin registration. It is required for plugin file to be in "luna_builder/rig_nodes" directory and have a name that begins with "node_". Besides that, a plugin file needs to have a definition for "register_plugin" function which will be used as entry point during initialization. Plugin has ability to register new data types, nodes and functions.

Data types are used to disallow invalid connections between nodes. In order to register new data type user would need to import "editor_conf" module and use "editor_conf.DataType.register_datatype" method which will need the following parameters for successful registration:

- Name in the register (can be accessed as attribute of editor_conf.DataType class)
- A class type associated with this data type
- Socket colour (QColor instance or hexadecimal code as string)
- Default label (optional)
- Default value (optional)

Node registration is done by calling "editor_conf.register_node" function and passing node ID and class as parameters. In case if ID is already taken an error will be reported during plugin load process. Currently implemented nodes are all rigging components, several utility nodes and constant value nodes.

Function registration is performed by calling "editor_conf.register"_function and passing the following parameters:

- Function reference.
- Source data type - used to link functions to data types to mimic method behaviour.
- Inputs dictionary – dictionary or OrderedDict of input name and data type pairs.
- Outputs dictionary – dictionary or OrderedDict of output name and data type pairs.
- Nice name – name that will be used as in node's palette and on node title on creation.
- Sub type – an additional string that will be added to function signature to allow the same function to be registered more than once.
- Category – subcategory in for node's palette
- Icon – file name of an icon located in "res/images/icons" directory.

If function was registered successfully, it will be available in node's palette just like a regular node. When dragged into the scene function will create a FunctionNode instance (Figure 22). It is a special node that is dynamically populated according to registration data. It holds a reference to an actual function, so during execution it will call stored function with input values passed as an argument list. The return of the function will be passed to the output sockets.

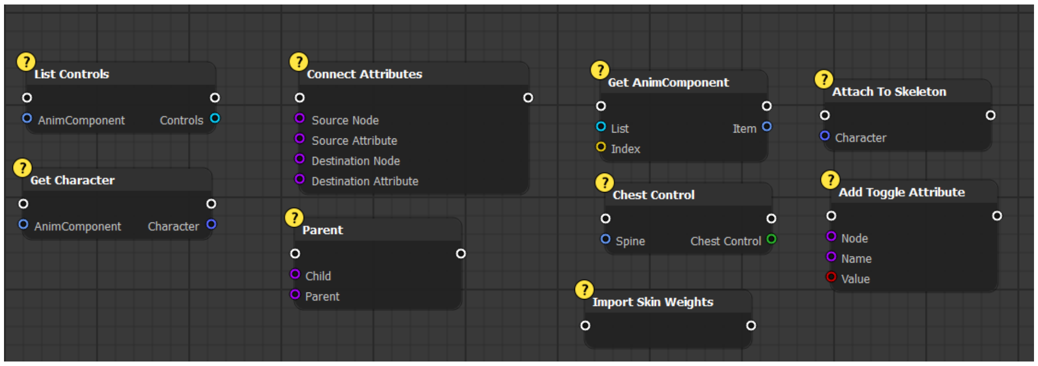
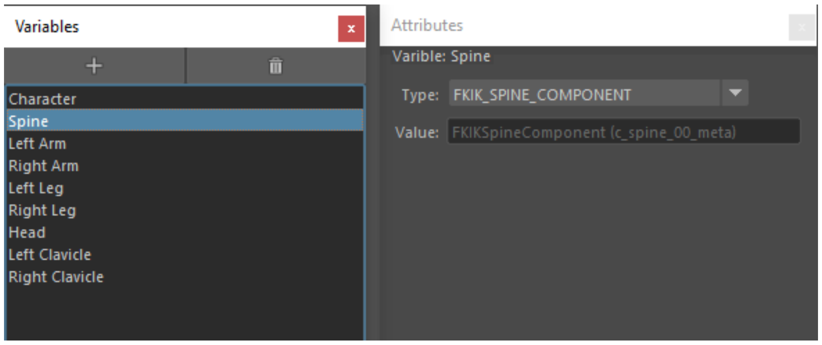


Figure 22: FunctionNode examples.

4.11.3 Scene variables

Early on in development this graph had a problem of extremely complicated node connections due to absence of a way to store node output value for later reuse. This was solved by introducing scene variables. Internally variables are represented by OrderedDict with pairs of variable name and tuple, consisting of value and data type name. User can create, delete and rename variables using "Variables" widget. Variable's type and value can be changed by selecting item in variables list and adjusting it in attributes editor (Figure 23). Value of the variable cannot be edited if data type of the variable is a subclass of a runtime data type, a list of those includes Component, list and Control. The same restriction applies to any input and output of the node.

Variable value can be set and retrieved during graph execution using get and set nodes. Unlike other node types, these nodes can only be instanced by dragging a variable from variable list into the graph scene (Figure 24). On drop of dragged list item user will have a choice to spawn either getter or setter node for dropped variable.


Variables

Character
Spine
Left Arm
Right Arm
Left Leg
Right Leg
Head
Left Clavicle
Right Clavicle

Attributes

Varible: Spine

Type: FKIK_SPINE_COMPONENT

Value: FKIKSpineComponent (c_spine_00_meta)

Figure 23: Scene variables and attributes widget.

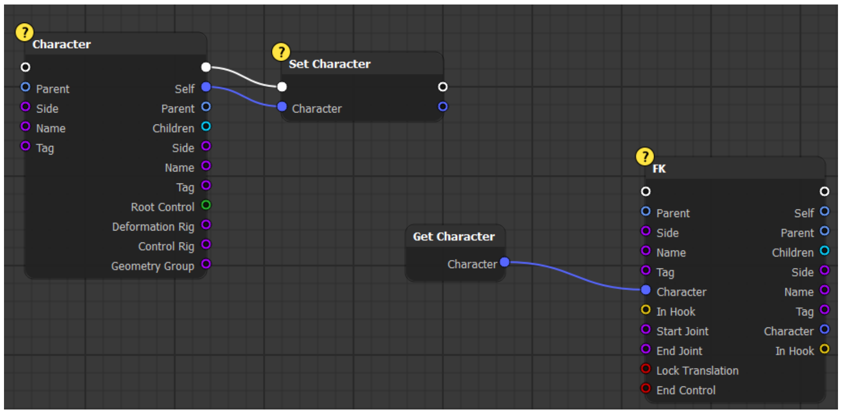


Figure 24: Get Set nodes.

4.12 Additional tools

To better illustrate the benefits of meta node based rigging a several minor tools were created. These tools use PySide2 library and are available from main Luna menu. The key concept that is shared between them is reusage of methods that are already implemented for components resulting in minimum code required for tool functionality.

4.12.1 Transfer keyframes tool

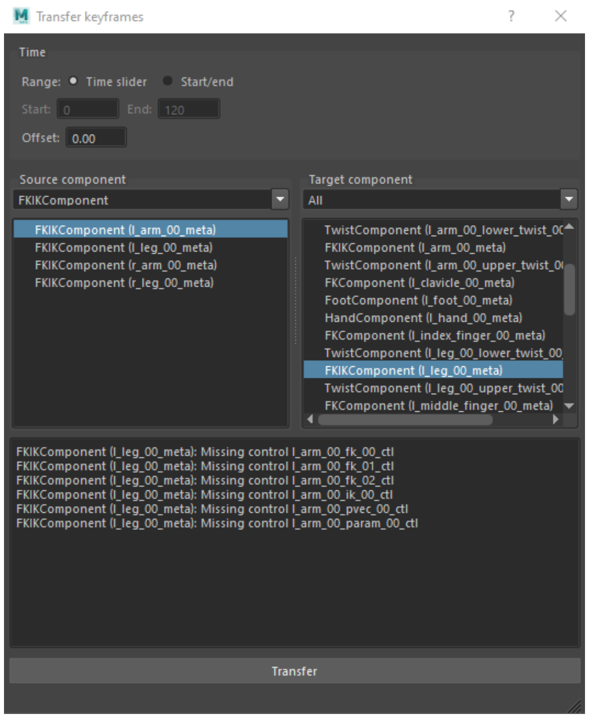
If user needs to copy animation keyframes from one rig to another it is a lot faster to do it using a transfer keyframes tool (Figure 25). The only thing that user will need to do is selecting source and target components and pressing "Transfer button". Similar to animation baker tool this will execute components Python method, although this time it is a "copy_keyframes" method.

4.12.2 Animation baker tool

This tool lets user to do animation baking operations with a rig. It has three operation modes: Skeleton, FKIK and Space (Figure 26). Skeleton mode is responsible for baking animation from rig to bind skeleton. It is useful for cases when animation needs to be exported to other package like a game engine. To bake an animation user needs to select components from the list and press "Bake to skeleton" button. This will run "bake_to_skeleton" method for each of selected components in the list.

FKIK baking mode operates on FKIK components enabling user to transfer animation between FK and IK controls. Bake result is comparable to Maya's HIK baking, the only addition will be ability to bake reverse foot rotation from IK to FK.

Space baking gives user an option to bake Control's animation from one space to another or to any transform in the scene.


Transfer keyframes
Time
Range: ● Time slider ○ Start/end
Start: 0 End: 120
Offset: 0.00
Source component
FKIKComponent
FKIKComponent (l_arm_00_meta)
FKIKComponent (l_leg_00_meta)
FKIKComponent (r_arm_00_meta)
FKIKComponent (r_leg_00_meta)
Target component
All
TwistComponent (l_arm_00_lower_twist_00
FKIKComponent (l_arm_00_meta)
TwistComponent (l_arm_00_upper_twist_0
FKComponent (l_clavicle_00_meta)
FootComponent (l_foot_00_meta)
HandComponent (l_hand_00_meta)
FKComponent (l_index_finger_00_meta)
TwistComponent (l_leg_00_lower_twist_00
FKIKComponent (l_leg_00_meta)
TwistComponent (l_leg_00_upper_twist_00
FKComponent (l_middle_finger_00_meta)
Transfer

```
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_fk_00_ctl
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_fk_01_ctl
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_fk_02_ctl
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_ik_00_ctl
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_pvec_00_ctl
FKIKComponent (l_leg_00_meta): Missing control l_arm_00_param_00_ctl
```

Figure 25: Transfer keyframes tool

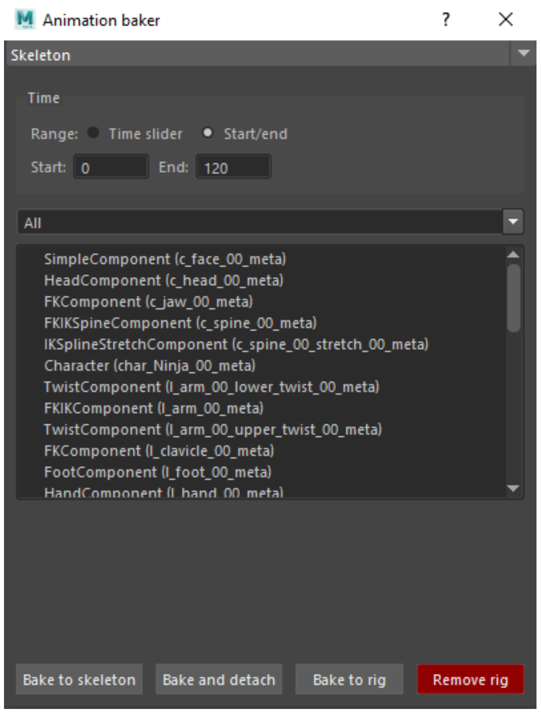

Animation baker
Skeleton
Time
Range: ○ Time slider ● Start/end
Start: 0 End: 120
All
SimpleComponent (c_face_00_meta)
HeadComponent (c_head_00_meta)
FKComponent (c_jaw_00_meta)
FKIKSpineComponent (c_spine_00_meta)
IKSplineStretchComponent (c_spine_00_stretch_00_meta)
Character (char_Ninja_00_meta)
TwistComponent (l_arm_00_lower_twist_00_meta)
FKIKComponent (l_arm_00_meta)
TwistComponent (l_arm_00_upper_twist_00_meta)
FKComponent (l_clavicle_00_meta)
FootComponent (l_foot_00_meta)
HandComponent (l_hand_00_meta)
Bake to skeleton
Bake and detach
Bake to rig
Remove rig

Figure 26: Animation baker (Skeleton mode)

4.12.3 SDK Pose exporter

Driven pose exporter (Figure 27) adds an easy option to create driven key poses and keep them between build iterations. An example where it can be particularly useful is creation of finger poses. These are the steps required to export fist pose for a hand with five fingers:

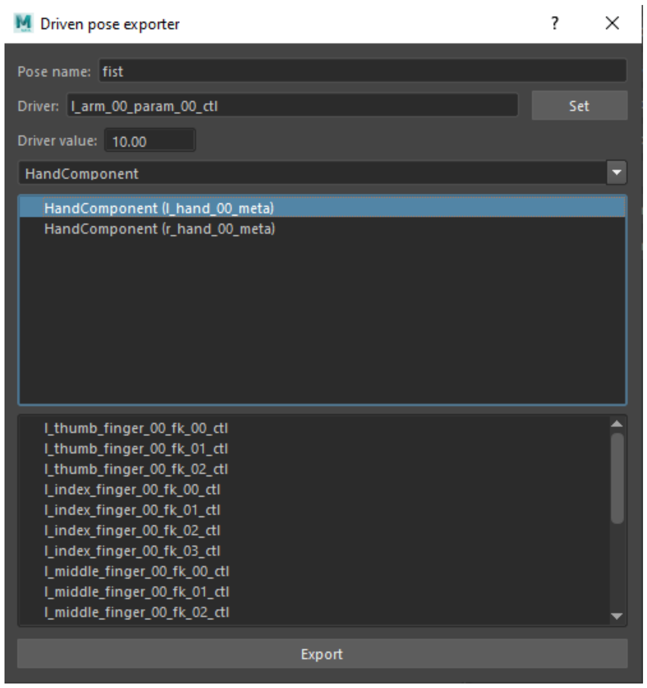
- Pose controls as desired
- Enter a pose name (will be used to create an attribute on the driver node)
- Either enter a driver's name manually or select object in the scene and press "Set".
- Enter driver's attribute value that will correspond to the pose.
- Select a component to export pose for
- Select controls to include into the pose (selecting none will include all the controls)
- Press export

- Use DrivenPoseManager.import_all to import poses during build.
-

When poses are imported there will be an offset node created per control that has a driven key connections.

4.12.4 SDK Corrective exporter

Corrective exporter (Figure 28) operates similar to Driven pose exporter, although it only exports poses for components of type CorrectiveComponent and does not create any additional attributes on the driver node.


Driven pose exporter

Pose name: fist

Driver: l_arm_00_param_00_ctl

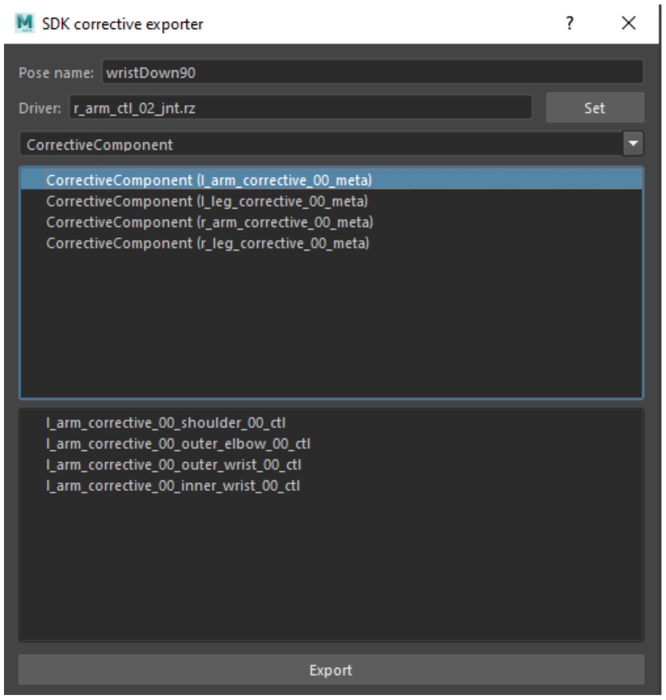
Driver value: 10.00

HandComponent

HandComponent (l_hand_00_meta)
HandComponent (r_hand_00_meta)

l_thumb_finger_00_fk_00_ctl
l_thumb_finger_00_fk_01_ctl
l_thumb_finger_00_fk_02_ctl
l_index_finger_00_fk_00_ctl
l_index_finger_00_fk_01_ctl
l_index_finger_00_fk_02_ctl
l_index_finger_00_fk_03_ctl
l_middle_finger_00_fk_00_ctl
l_middle_finger_00_fk_01_ctl
l_middle_finger_00_fk_02_ctl

Figure 27: SDK Pose exporter


SDK corrective exporter

Pose name: wristDown90

Driver: r_arm_ctl_02_jnt.rz

CorrectiveComponent

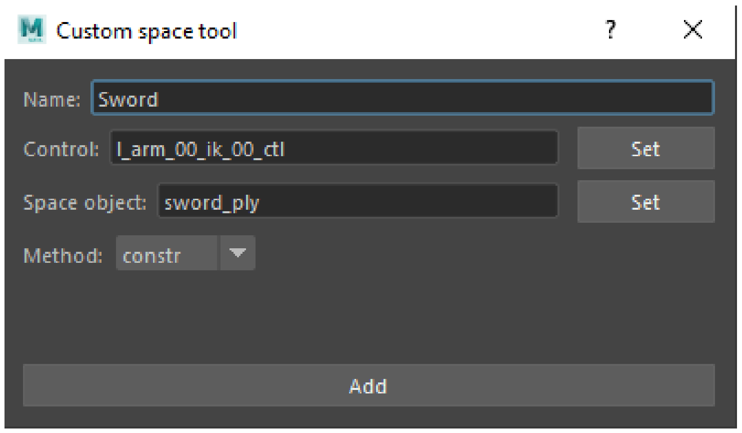
CorrectiveComponent (l_arm_corrective_00_meta)
CorrectiveComponent (l_leg_corrective_00_meta)
CorrectiveComponent (r_arm_corrective_00_meta)
CorrectiveComponent (r_leg_corrective_00_meta)

l_arm_corrective_00_shoulder_00_ctl
l_arm_corrective_00_outer_elbow_00_ctl
l_arm_corrective_00_outer_wrist_00_ctl
l_arm_corrective_00_inner_wrist_00_ctl

Figure 28: SDK Corrective exporter.

4.12.5 Custom space tool

This tool is mainly targeted towards animators allowing them to add custom spaces to controls during animation process. The only parameters required for that are: space name, control to add space to and a space object. This tool does not implement any space adding logic, instead it uses an already defined "add_space" method of a Control class.


Custom space tool

Name: Sword

Control: l_arm_00_ik_00_ctl

Space object: sword_ply

Method: constr

Figure 29: Custom space tool.

5 Conclusion

The resulting system ended up being light weight and easily extendable, but some parts of it could benefit from refactoring for further reduction of code reuse and general optimizations.

System was mainly tested on Maya running Python 2.7.11, and despite code being written with Python 3 in mind, there are still some features that do not work as intended in Maya 2022 due to Maya's internal changes. Node editor requires optimization as performance of large scenes is significantly lower and in most cases that's what most rig builds will end up as.

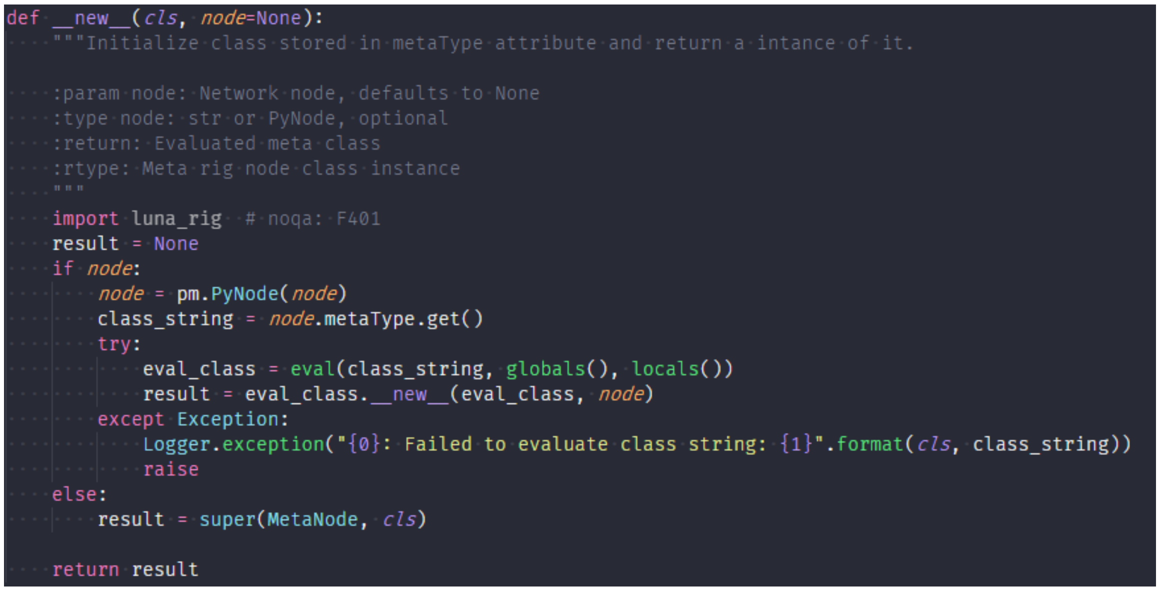
In its current state the pipeline is limited to use withing Maya, but it is possible to implement it as a standalone system that will have abstract definitions of rig components and DCC plugins that will do the concrete implementation. This way the system will be decoupled from DCC's python version and will be available for a much bigger userbase.

6 Bibliography

Animation Studios, n.d. *Advanced Skeleton.* [Online]
Available at: https://www.animationstudios.com.au/advanced-skeleton
[Accessed 25 July 2021].
Autodesk, 2017. *Maya API introduction.* [Online]
Available at: https://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=__files_API_Introduction_htm
[Accessed 25 July 2021].
Campos, M., Villar, R., Jerome, D. & Lesage, C., 2021. *mGear Framework.* [Online]
Available at: http://www.mgear-framework.com/
[Accessed 25 July 2021].
Epic Games, n.d. *Unreal Engine documentation - Nodes.* [Online]
Available at: https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Nodes/
[Accessed 23 July 2021].
GeekForGeeks, 2019. *__new__ in Python.* [Online]
Available at: https://www.geeksforgeeks.org/__new__-in-python/
[Accessed 20 July 2021].
Hunt, D., 2009. *Modular Procedural Rigging GDC, BUNGIE.* [Online]
Available at: https://dokumen.tips/documents/modular-procedural-rigging-gdc-2009-david-hunt-bungie.html
[Accessed 10 July 2021].
Hunt, D. & Forrest, S., 2015. *Tools-Based Rigging in Bungie's Destiny.* [Online]
Available at: https://www.gdcvault.com/play/1022339/Tools-Based-Rigging-in-Bungie
[Accessed 25 June 2021].
Shotarov, V., 2017. *Maya matrix nodes - Part 1: Node based matrix constraint.* [Online]
Available at: https://bindpose.com/maya-matrix-based-functions-part-1-node-based-matrix-constraint/
[Accessed 28 June 2021].
TechTerms, 2021. *Metadata definition.* [Online]
Available at: https://techterms.com/definition/metadata
[Accessed 24 July 2021].

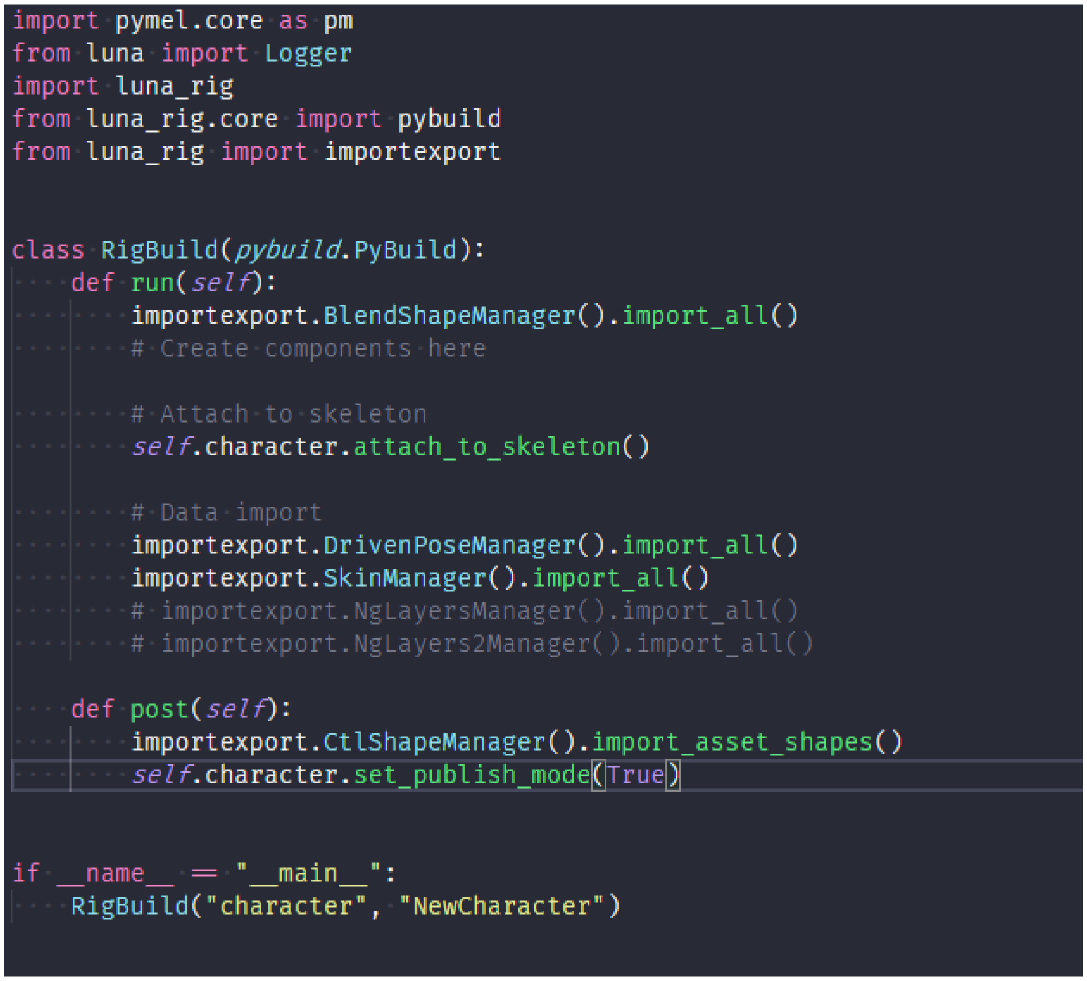
7 Appendix A

Code snippets



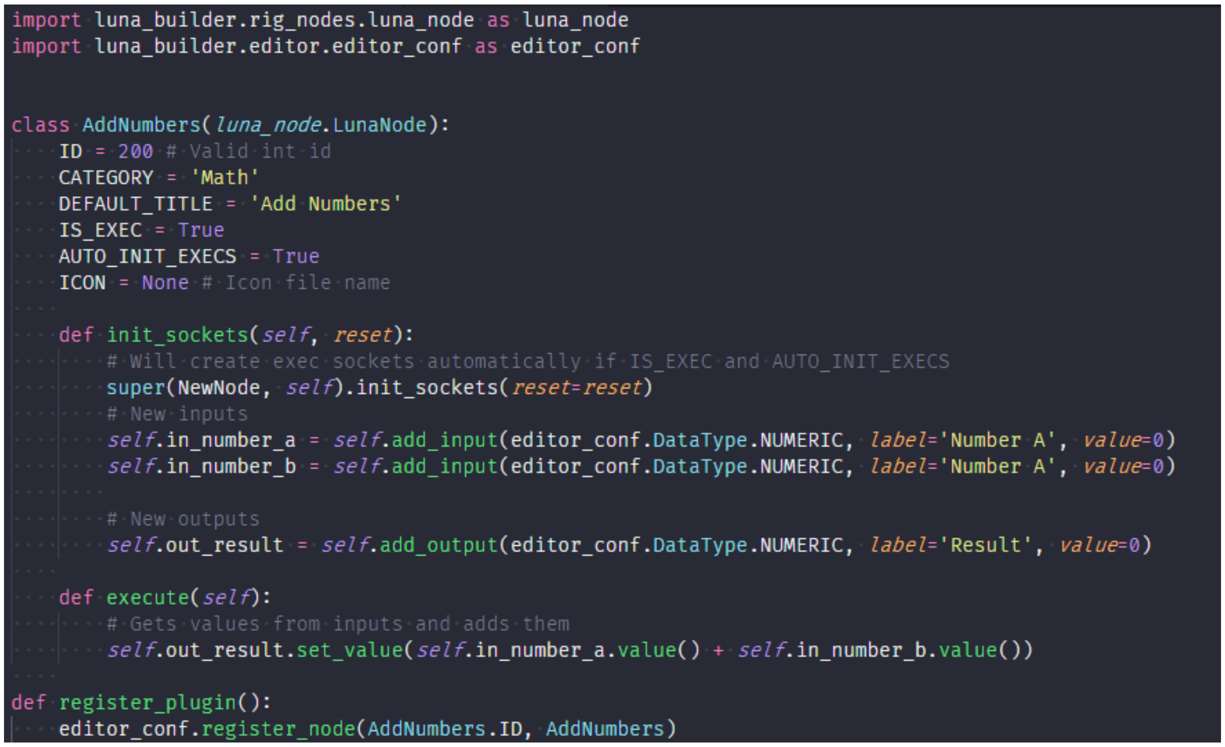
```python
def __new__(cls, node=None):
    """Initialize class stored in metaType attribute and return a intance of it.

    :param node: Network node, defaults to None
    :type node: str or PyNode, optional
    :return: Evaluated meta class
    :rtype: Meta rig node class instance
    """
    import luna_rig  # noqa: F401
    result = None
    if node:
        node = pm.PyNode(node)
        class_string = node.metaType.get()
        try:
            eval_class = eval(class_string, globals(), locals())
            result = eval_class.__new__(eval_class, node)
        except Exception:
            Logger.exception("{0}: Failed to evaluate class string: {1}".format(cls, class_string))
            raise
    else:
        result = super(MetaNode, cls)

    return result
```

Figure 30: MetaNode __new__ method code.



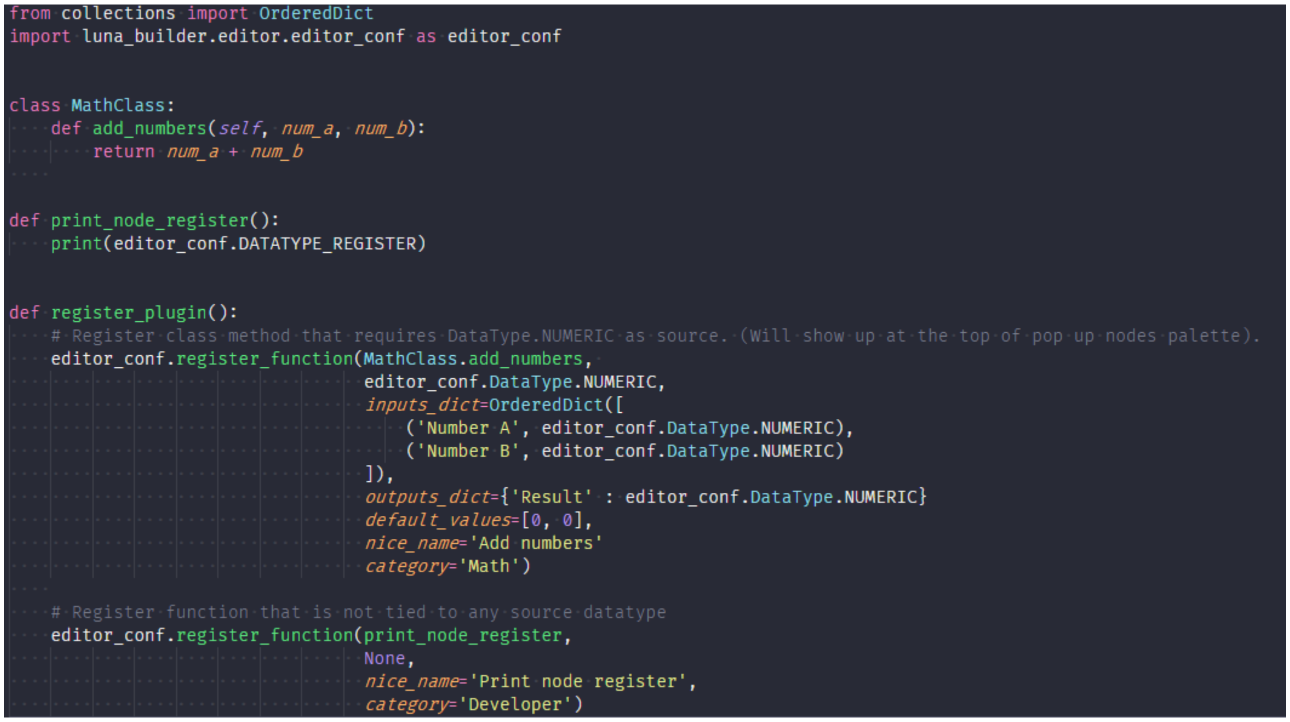
```python
import pymel.core as pm
from luna import Logger
import luna_rig
from luna_rig.core import pybuild
from luna_rig import importexport


class RigBuild(pybuild.PyBuild):
    def run(self):
        importexport.BlendShapeManager().import_all()
        # Create components here

        # Attach to skeleton
        self.character.attach_to_skeleton()

        # Data import
        importexport.DrivenPoseManager().import_all()
        importexport.SkinManager().import_all()
        # importexport.NgLayersManager().import_all()
        # importexport.NgLayers2Manager().import_all()

    def post(self):
        importexport.CtlShapeManager().import_asset_shapes()
        self.character.set_publish_mode(True)


if __name__ == "__main__":
    RigBuild("character", "NewCharacter")
```

Figure 31: PyBuild build template.



```python
import luna_builder.rig_nodes.luna_node as luna_node
import luna_builder.editor.editor_conf as editor_conf


class AddNumbers(luna_node.LunaNode):
    ID = 200 # Valid int id
    CATEGORY = 'Math'
    DEFAULT_TITLE = 'Add Numbers'
    IS_EXEC = True
    AUTO_INIT_EXECS = True
    ICON = None # Icon file name

    def init_sockets(self, reset):
        # Will create exec sockets automatically if IS_EXEC and AUTO_INIT_EXECS
        super(NewNode, self).init_sockets(reset=reset)
        # New inputs
        self.in_number_a = self.add_input(editor_conf.DataType.NUMERIC, label='Number A', value=0)
        self.in_number_b = self.add_input(editor_conf.DataType.NUMERIC, label='Number A', value=0)

        # New outputs
        self.out_result = self.add_output(editor_conf.DataType.NUMERIC, label='Result', value=0)

    def execute(self):
        # Gets values from inputs and adds them
        self.out_result.set_value(self.in_number_a.value() + self.in_number_b.value())

def register_plugin():
    editor_conf.register_node(AddNumbers.ID, AddNumbers)
```

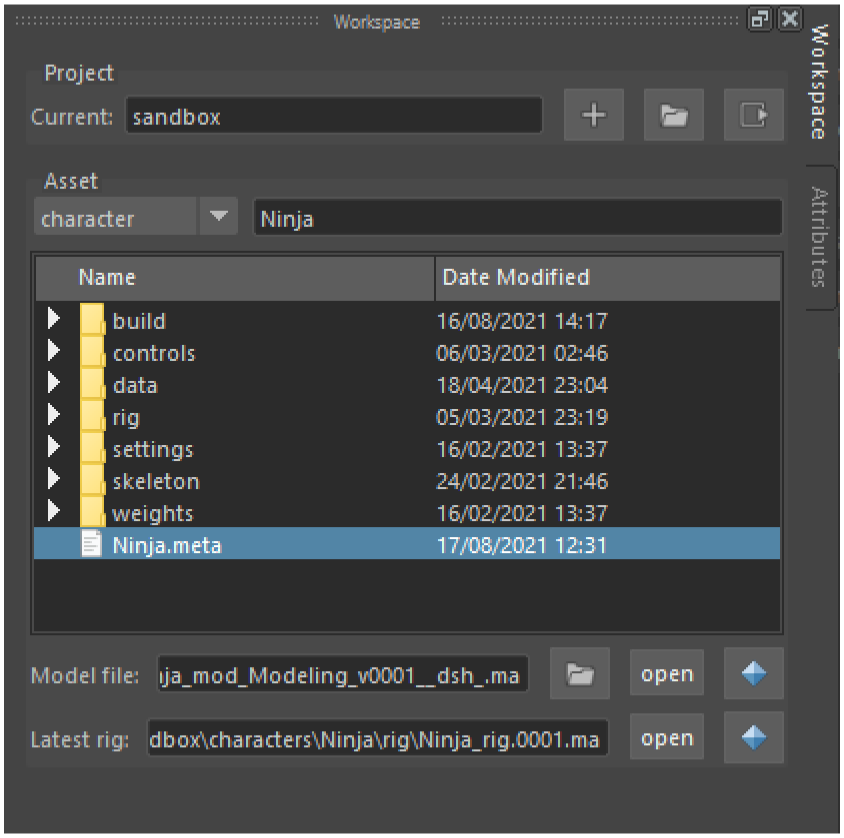
Figure 32: Custom node plugin example.



```python
from collections import OrderedDict
import luna_builder.editor.editor_conf as editor_conf


class MathClass:
    def add_numbers(self, num_a, num_b):
        return num_a + num_b


def print_node_register():
    print(editor_conf.DATATYPE_REGISTER)


def register_plugin():
    # Register class method that requires DataType.NUMERIC as source. (Will show up at the top of pop up nodes palette).
    editor_conf.register_function(MathClass.add_numbers,
                                  editor_conf.DataType.NUMERIC,
                                  inputs_dict=OrderedDict([
                                      ('Number A', editor_conf.DataType.NUMERIC),
                                      ('Number B', editor_conf.DataType.NUMERIC)
                                  ]),
                                  outputs_dict={'Result' : editor_conf.DataType.NUMERIC}
                                  default_values=[0, 0],
                                  nice_name='Add numbers'
                                  category='Math')

    # Register function that is not tied to any source datatype
    editor_conf.register_function(print_node_register,
                                  None,
                                  nice_name='Print node register',
                                  category='Developer')
```

Figure 33: Function registration example.

Luna builder interface elements



Project

Current: sandbox

Asset

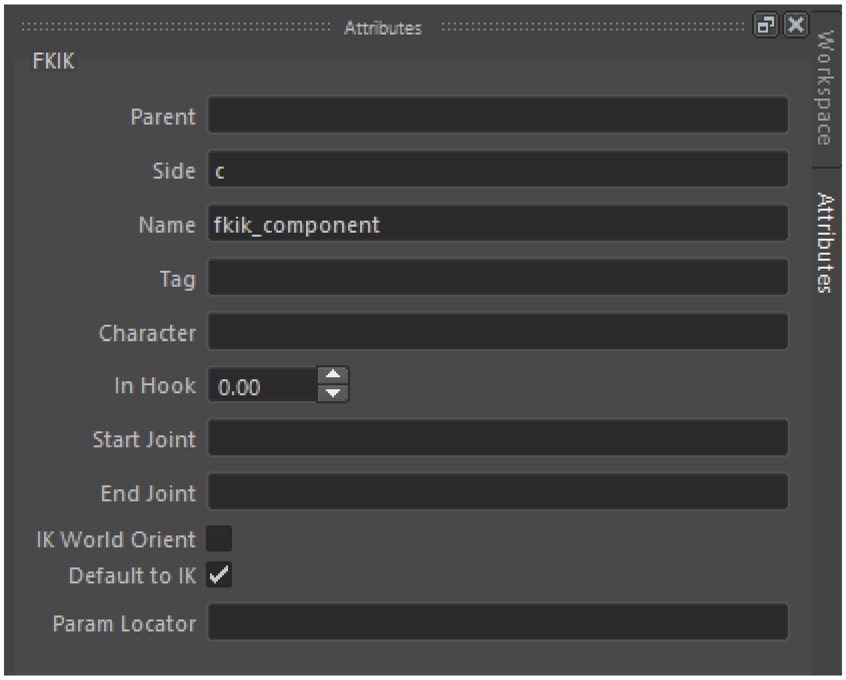
character | Ninja

Name	Date Modified
build	16/08/2021 14:17
controls	06/03/2021 02:46
data	18/04/2021 23:04
rig	05/03/2021 23:19
settings	16/02/2021 13:37
skeleton	24/02/2021 21:46
weights	16/02/2021 13:37
Ninja.meta	17/08/2021 12:31

Model file: ıja_mod_Modeling_v0001__dsh_.ma open

Latest rig: dbox\characters\Ninja\rig\Ninja_rig.0001.ma open

Figure 34: Workspace widget.



FKIK

Parent	
Side	c
Name	fkik_component
Tag	
Character	
In Hook	0.00
Start Joint	
End Joint	
IK World Orient	
Default to IK	✓
Param Locator	

Figure 35: Attributes widget (FKIKComponent node selected).

Available editor nodes

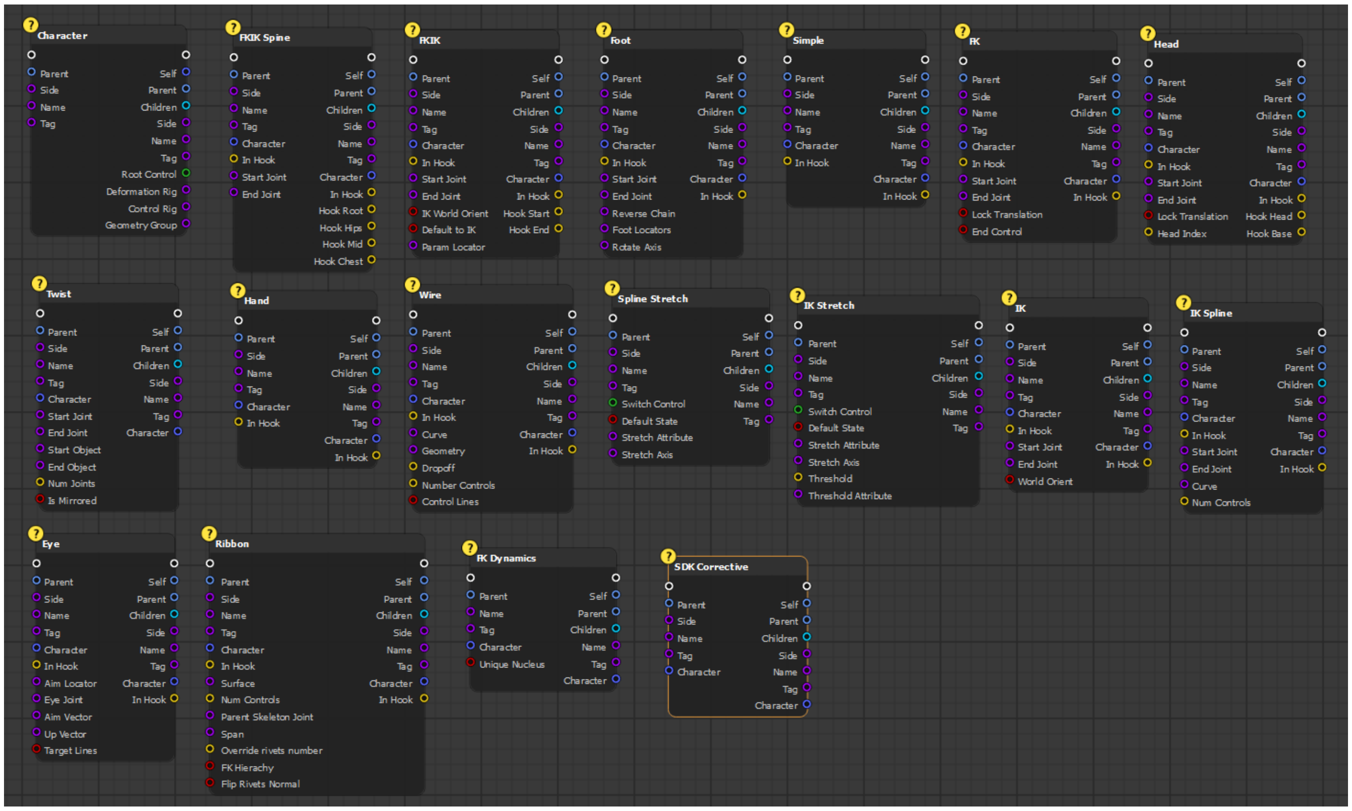


Figure 36: Available rig component nodes.

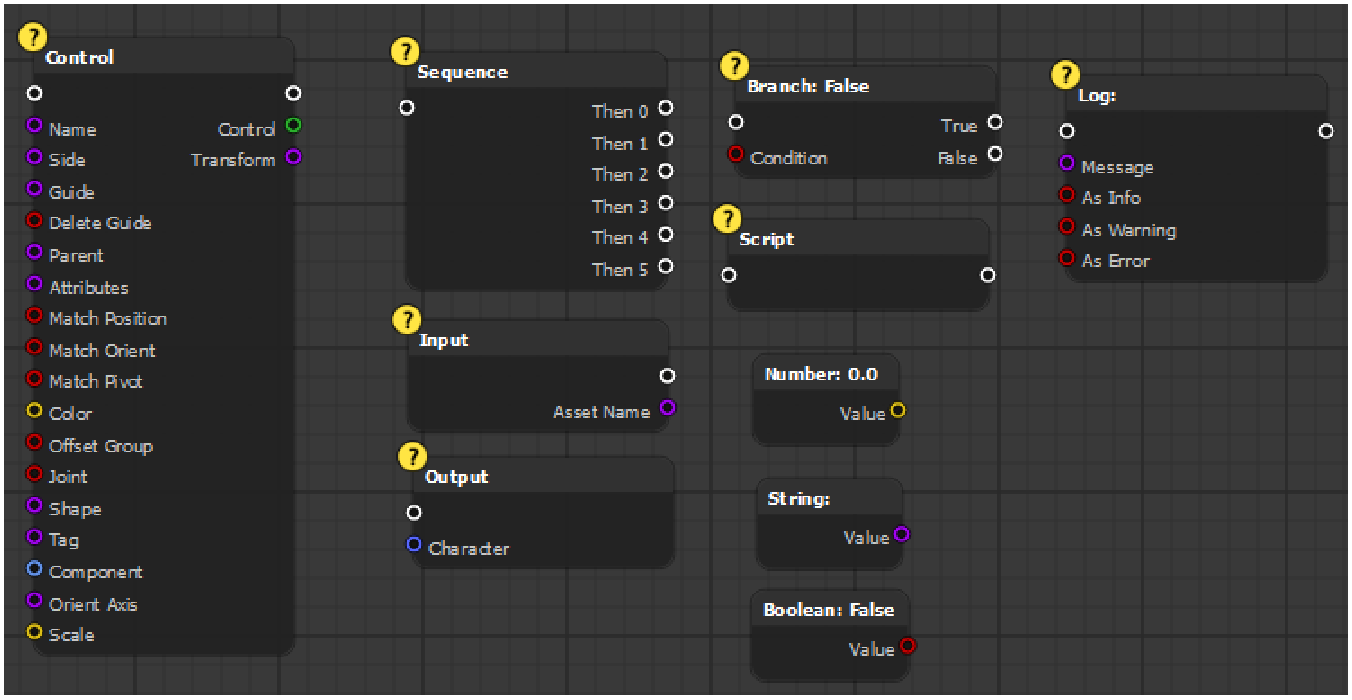
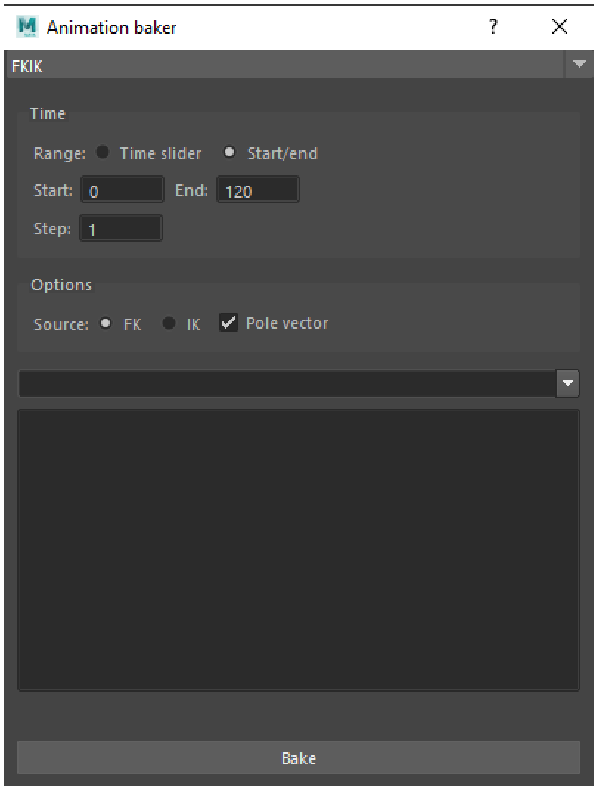


Figure 37: Available utility nodes.

Alternative animation baker modes


Animation baker ? ×

FKIK

Time

Range: ● Time slider ● Start/end

Start: 0 End: 120

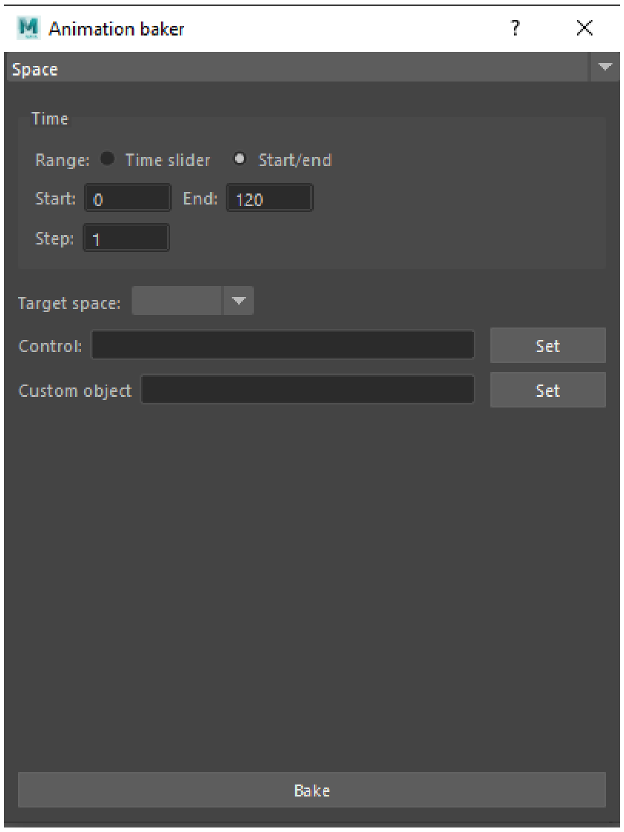
Step: 1

Options

Source: ● FK ● IK ✓ Pole vector

Bake

Figure 38: Animation baker FKIK mode.


Animation baker ? ×

Space

Time

Range: ● Time slider ● Start/end

Start: 0 End: 120

Step: 1

Target space:

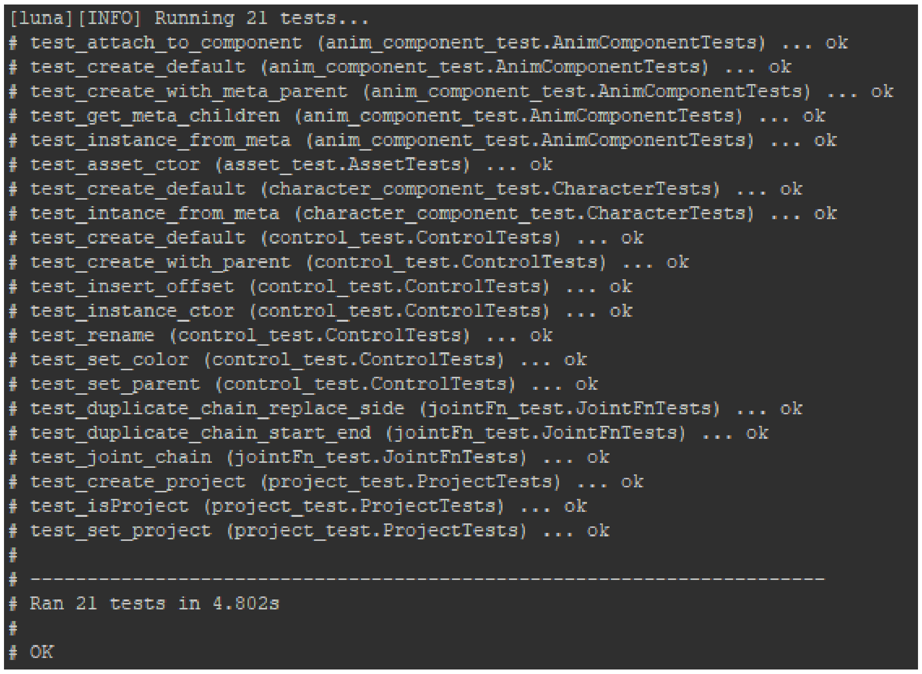
Control: Set

Custom object Set

Bake

Figure 39: Animation baker space mode.

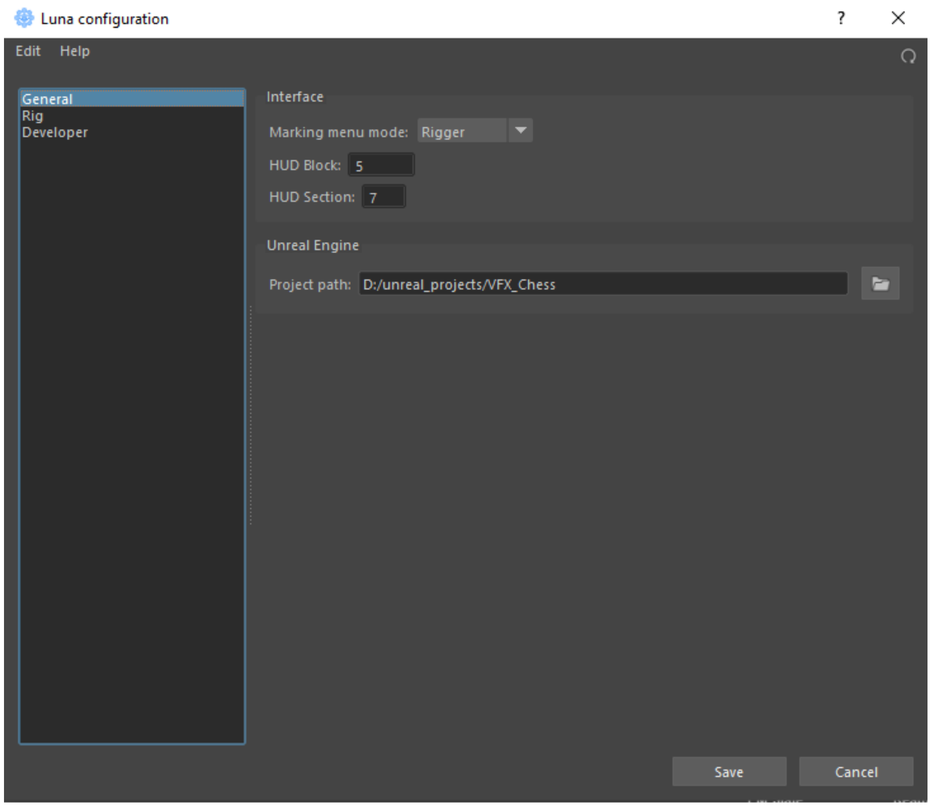
Miscellaneous features:



```
[luna][INFO] Running 21 tests...
# test_attach_to_component (anim_component_test.AnimComponentTests) ... ok
# test_create_default (anim_component_test.AnimComponentTests) ... ok
# test_create_with_meta_parent (anim_component_test.AnimComponentTests) ... ok
# test_get_meta_children (anim_component_test.AnimComponentTests) ... ok
# test_instance_from_meta (anim_component_test.AnimComponentTests) ... ok
# test_asset_ctor (asset_test.AssetTests) ... ok
# test_create_default (character_component_test.CharacterTests) ... ok
# test_intance_from_meta (character_component_test.CharacterTests) ... ok
# test_create_default (control_test.ControlTests) ... ok
# test_create_with_parent (control_test.ControlTests) ... ok
# test_insert_offset (control_test.ControlTests) ... ok
# test_instance_ctor (control_test.ControlTests) ... ok
# test_rename (control_test.ControlTests) ... ok
# test_set_color (control_test.ControlTests) ... ok
# test_set_parent (control_test.ControlTests) ... ok
# test_duplicate_chain_replace_side (jointFn_test.JointFnTests) ... ok
# test_duplicate_chain_start_end (jointFn_test.JointFnTests) ... ok
# test_joint_chain (jointFn_test.JointFnTests) ... ok
# test_create_project (project_test.ProjectTests) ... ok
# test_isProject (project_test.ProjectTests) ... ok
# test_set_project (project_test.ProjectTests) ... ok
#
# ----------------------------------------------------------------------
# Ran 21 tests in 4.802s
#
# OK
```

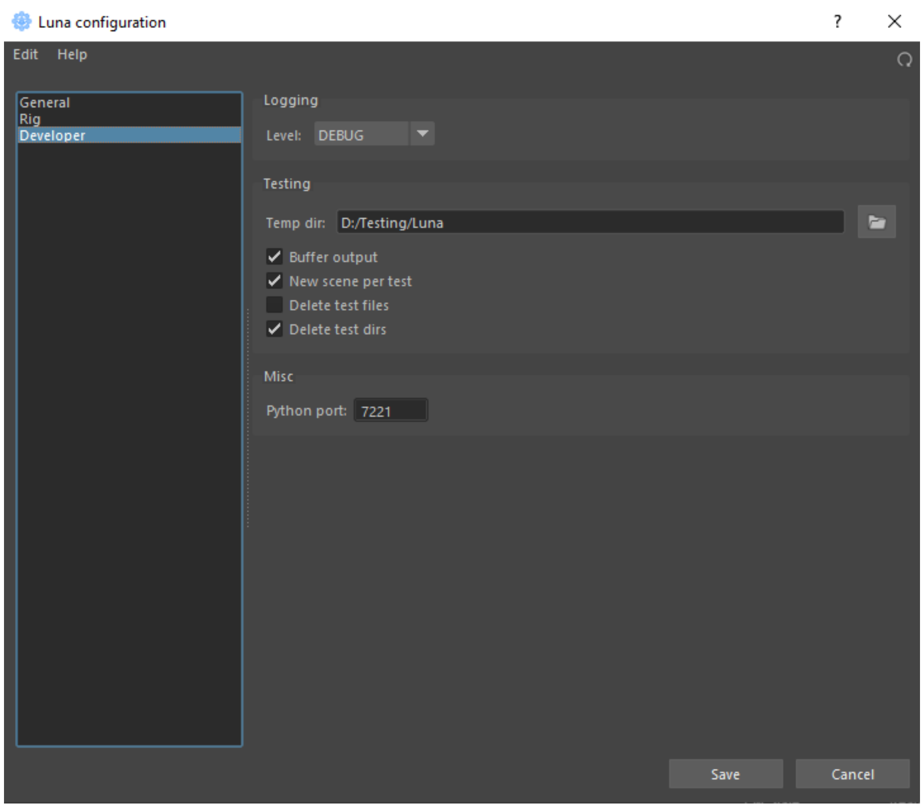
Figure 40: Implemented pipeline unit testing.

Configuration dialog categories



Marking menu mode: Rigger
HUD Block: 5
HUD Section: 7
Project path: D:/unreal_projects/VFX_Chess

Figure 41: Configuration dialog (General category).



Logging

Level: DEBUG

Testing

Temp dir: D:/Testing/Luna

- [x] Buffer output
- [x] New scene per test
- [ ] Delete test files
- [x] Delete test dirs

Misc

Python port: 7221

Figure 42: Configuration dialog (Developer category).