A Real-Time Dynamic Simulator and an Associated Front-End Representation Format for Simulating Complex Robots and Environments

Conference Paper · November 2019

DOI: 10.1109/IROS40897.2019.8968568

CITATIONS
27

READS
317

4 authors, including:

Yan Wang
The Chinese University of Hong Kong
7 PUBLICATIONS **78** CITATIONS
SEE PROFILE

Radian Gondokaryono
University of Toronto
7 PUBLICATIONS **114** CITATIONS
SEE PROFILE

Gregory S Fischer
Worcester Polytechnic Institute
163 PUBLICATIONS **4,100** CITATIONS
SEE PROFILE

2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Macau, China, November 4-8, 2019

A Real-Time Dynamic Simulator and an Associated Front-End Representation Format for Simulating Complex Robots and Environments

Adnan Munawar, Yan Wang, Radian Gondokaryono and Gregory S. Fischer

Abstract— **Robot Dynamic Simulators offer convenient implementation and testing of physical robots, thus accelerating research and development. While existing simulators support most real-world robots with serially linked kinematic and dynamic chains, they offer limited or conditional support for complex closed-loop robots. On the other hand, many of the underlying physics computation libraries that these simulators employ support closed-loop kinematic chains and redundant mechanisms. Such mechanisms are often utilized in surgical robots to achieve constrained motions (e.g., the remote center of motion (RCM)). To deal with such robots, we propose a new simulation framework based on a front-end description format and a robust real-time dynamic simulator. Although this study focuses on surgical robots, the proposed format and simulator are applicable to any type of robot. In this manuscript, we describe the philosophy and implementation of the front-end description format and demonstrate its performance and the simulator's capabilities using simulated models of real-world surgical robots.**

I. INTRODUCTION

The Universal Robot Description Format (URDF) is one of the most widely used representation formats for robots. There are other formats that are either driven from URDF (Standard Description Format (SDF) [1]) or allow conversion from URDF (such as MuJoCo [2] format and V-REP Simulator [3]). Arguably, URDF played a pivotal role in the success and community adoption of Robot Operating System (ROS) [4] and is tailored to serial manipulators and robots. While there are ways to visually achieve redundant mechanisms using **mimic** tags, realistic closed-loop constraints are not possible as the limitation broadly comes from the design philosophy of URDF. The idea of a robot, as envisioned by URDF, is a spatial tree of bodies wherein the joints are essential parts of the links. While this philosophy is the foundational building block of kinematics and visualizations using the default ROS simulator RViz (and derivative software such as MoveIt), it thwarts the ability to define unconnected, sparsely and densely connected combination of bodies.

The Simulation Description Format (SDF) is employed by the Gazebo Simulator [1] and is similar to the URDF in many core aspects while defining serial robots. SDF addresses the key limitation of the URDF in defining closed-loop mechanisms, however, the latest Gazebo simulator (9.0) does not support direct control of parallel linkages using ROS. While URDF can only define a single robot per description file, SDF has the capacity to support the distributed description of robots. Moreover, SDF is designed for more general purpose use with support for environment entities such as lighting, scene-objects, and sensors. For the purpose of this discussion, we shall limit the role of the current robot description formats in their capability to represent kinematic/dynamic objects and not necessarily environmental/scene objects.

Both the URDF and SDF (and even MuJoCo) use XML language, which although historically has been used to store and transmit configuration and description data, is not known for human readability. This limitation has somewhat been the reason behind the development of other markup languages such as JavaScript Object Notation (JSON) `https://www.json.org/` and Yet Another Markup Language (YAML) `https://yaml.org/` . While XML retains its place as the back-end tool for data storage, YAML and JSON are gaining wide adoptability in front-end applications. In addition to the readability component, both JSON and YAML are feature-rich as compared to XML. For example, YAML provides inherent support for macros in the form of **anchors**, which tend to be useful for specification of properties.

Gazebo [1] is supported across major operating systems (e.g., Microsoft Windows, Mac OS, and Linux), however, it is used most commonly with ROS (Linux). While Gazebo is feature-rich and allows for robust support for a large number of sensors as loadable plugins, its support with URDF, and consequently external control via ROS-topics is complicated and non-robust. In fact, the process of going from a URDF to SDF, and eventually loading joint controllers, communicable using ros-topics/ros-services, is lengthy and repetitive even for advanced users. Some of this complexity can be attributed to **ros_control** and **ros_controllers** packages which form the backbone of control via ROS. Even after a successful bridge between ROS and Gazebo has been established, joint control for connected bodies requires extra steps since the joints must be controlled independently using messages and services. There are of course ways to simplify the segregation of joint controllers by using wrappers, such as the Gazebo plug-in for da Vinci Surgical Robot [5]. While this might not pose an issue for simpler robots with a limited number of joints, it creates unnecessary complexity for real-world surgical robots.

Adnan Munawar, Yan Wang, Radian Gondokaryono and Gregory S. Fischer are with the Robotic Engineering Department, Worcester Polytechnic Institute, WPI, 100 Institute Road, MA, USA amunawar@wpi.edu[1]

This work is supported by the National Science Foundation through National Robotics Initiative (NRI) grant: **IIS-1637759**

II. The AMBF Format

Based on the limitations of the robot description formats, and consequently robot simulators elaborated in Section I the following metrics are outlined for the proposed Asynchronous Multi-Body Framework Format (AMBF Format):

- **Human Readability**: One of the design motivations behind the Asynchronous Multi-Body Framework Format (AMBF Format or AMBF description file) is human readability, and consequently modification by hand. AMBF Format's design philosophy places robot description at the front-end for creating, modifying and distributed testing of multi-bodies.
- **Distributed Definition**: All the relevant data for a single body/constraint/environmental object should be contained in the relevant definition block. Removal of the data block should not affect any other body/constraint.
- **Constraint Handling**: A body could have multiple constraints (joints), and each constraint is defined independently of other constraints. Addition/removal of a constraint should not alter any other constraint except for the physical/dynamic implications.
- **Controllability**: In this context, controllability refers to the ability to apply forces on the body internally or externally from the running simulation independent of the other bodies. The connected bodies react passively based on the type of constraint they share.
- **Communicability**: This refers to the ability to relay information about all aspects of every body independent from each other. This information can include the constraints this body forms with all of its connected bodies but not necessarily the information of bodies themselves.
- **Dynamic Loading**: This defines the ability to add bodies at run-time and even define constraints between newly added bodies with existing bodies.

A. Anatomy of AMBF Format

The AMBF simulator was designed around the AMBF format to demonstrate its capabilities. The AMBF simulator uses several external packages that include Bullet Physics [6] and CHAI-3D [7]. The types of data in the AMBF format can be separated into various different types that include World Data, Rigid Body Data, Soft Body Data, Constraint Data, Lighting Data, Camera (View-port) Data and Input Device Data. The flexibility of the AMBF format allow not only for the definition of multiple robots and multi-bodies in one description file, but also for the separation of a single robot/multi-body in multiple description files, which is in line with the *Distributed Definition* metric. As an implementation example, all the body data for one robot can be defined in one or more description files, whereas the constraint (joint) data can be placed in separate file(s). The AMBF description files are written based on the AMBF Format. Figure 1 outlines the components of the AMBF description file (placed in tiles for emphasis but are written sequentially). The contents of the yellow tile are placed at the top and consist of global parameters applicable to the rest of the description file. Debugging

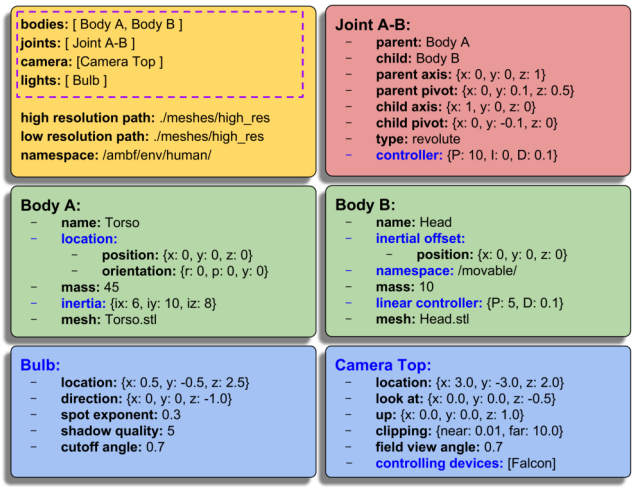


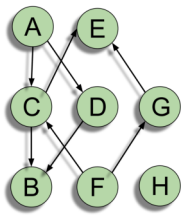
Fig. 1: The anatomy of AMBF format. The yellow tile forms the header and consists of global parameters and header lists which are highlighted with the purple dotted border. The red tile represents a constraint, green represents bodies and blue represents scene objects. The blue text highlights optional parameters.

robot/multi-body models by ignoring certain sub-components of the model is often an overlooked and understated design feature of robot description formats. Commenting out parts of the robot description is helpful, not only for debugging, but also for testing sub-components of a model in isolation. To ignore certain objects from loading in URDF or SDF, the required object's description spanning several lines needs to be commented out. AMBF's design specification uses header lists (emphasized by the dotted purple border in the yellow tile in Figure 1). The header lists are the entry point of the document such that bodies, visual elements and constraints are processed based on the content of these lists. Instead of having to comment out multiple lines of object data, it is sufficient to remove the object from header list of its type. The ignored description block does not affect the loading of any other body or constraint since the AMBF simulator, it's derivatives and the AMBF format are implemented while considering the *Distributed Definition*, *Constraint Handling* and *Dynamic Loading* specifications.

B. Densely Connected Tree of Bodies

To generate non-connected, semi-connected or densely connected bodies, we employ a combination of a graph network and a densely interconnected tree structure. Figure 2 illustrates an example of this composite structure. Unlike other formats where the parent refers to the immediate predecessor body, we relax this limitation by classifying all the predecessors of the body as its parents. While the relaxation of such parent hierarchy might seem counterintuitive in traditional robot representation formats, this relaxation is essential to meet the defined metrics of AMBF.

The Table in Figure 2 shows the resulting population of each body's lineage. It should be noted that the lineage path from one body to another may lead from multiple routes as



Bodies	Parents	Children
A	-	C, E, D, B
B	D, A, C, F	-
C	A, F	B
D	A	B
E	C, A, G, F	-
F	-	C, E, G
G	F	E
H	-	-

Fig. 2: Densely connected bodies with the corresponding lineage for each body shown on the right.

is the case between bodies ($A \rightarrow B$) and ($F \rightarrow E$). In such cases, we restrict adding children/parents redundantly to a body's lineage. To achieve the fully connected tree, we use an upward and a downward pass for each added constraint. Algorithm 1 sums the process of adding a constraint. At the end of all passes, each object maintains references to all the successor joints while all the children register references to all the predecessor bodies. It's important to note that in the case of diverging leaf nodes at a specific body, the predecessor bodies contain the references to all the children in every leaf node, however, the successors in leaf nodes are unaware of bodies in other leaves.

Algorithm 1 Add Constraint Algorithm

function ADD JOINT($joint, parent, child$)
  $p := parent$, $c := child$
  $c.Parents \leftarrow p$, $p.Children \leftarrow c$, $p.Joints \leftarrow joint$
  $UpwardTreePass(p)$
  $DownwardTreePass(c)$
end function
function UPWARD TREE PASS($body$)
  $P := body.Parents, C := body.Children$
  for $p \in P$ **do**
    for $c \in C$ **do**
      $p.Children \cup C$ & $p.Joints \cup c.Joints$
    end for
  end for
end function
function DOWNWARD TREE PASS($body$)
  $P := body.Parents, C := body.Children$
  for $c \in C$ **do**
    for $p \in P$ **do**
      $c.Parents \cup p.Parents$
    end for
  end for
end function

C. Convention of Constraint Definition

Constraints are used to connect two bodies in certain ways that limit their relative motion. In robot applications, the constraints can broadly be classified into two foundational types, the rotational constraint (revolute and hinge) and the translational constraint (prismatic and slider). Other constraints such as springs, cams, gears and 6 DOF joints can be built with the combination of foundational types. Fixed constraints present a special case, but can also be implemented with either of the foundational types.

In alignment with the design philosophy of the Asynchronous Framework, constraints are defined in a slightly different manner as compared to URDF or SDF. In URDF, the joint is treated at the origin of the child body or vice-versa, and two additional fields are used to set the offset of the child body's visual mesh and the collision mesh. Furthermore, these visual and collision offsets are defined in the body's definition, while the child origin (joint origin) is defined in the joint description. This distribution of data breaks the Asynchronous Design since, to get a complete specification of the interconnection between two bodies, it is necessary to parse the data beyond what is just defined in the constraint description.

In the AMBF Constraint definition, a body's origin is always treated as the base frame of its representative mesh. The way AMBF's constraint definition differs from URDF or SDF is by treating the constraint origin as independent of the child's or parent's body origin. As a result, two fields - namely pivot and axis - are used for the parent and the child. The pivot defines the location of the constraint from the body's origin in Cartesian space, and the axis defines the free axis in the body's frame. This convention requires less parameters to fully define an interconnection (13 = [parent's pivot (3) + parent's axis (3) + child's pivot (3) + child's axis (3) + offset (1)]) as compared to URDF or SDF (15 = [joint's XYZ (3) + joints's RPY (3) + joints's axis (3) + child's offset XYZ (3) + child's offset RPY (3)]). While this description is sufficient in constraining the two bodies, an extra scalar parameter is required to define the rotational offset along the parent axis between the two bodies. This offset is discussed in detail in section III-B.

The direct use of parent/child axes to build constraints emphasizes the front-end nature of AMBF, which consequently makes the specification of robots and multi-bodies easier. This, however, adds more work at the back-end where the constraints are actually parsed and processed. Internally, joint transforms w.r.t. the parent body and child transforms w.r.t. to the joint have to be computed. A unified convention to define the rotation represented by axes in the parent/child body frame is required. This convention utilizes the plane formed by the two axes ($a\vec{x}_p$ and $a\vec{x}_c$) to define a rotation matrix. This rotation is trivial except for the case where the two axes are parallel to each other since there exist an infinite number of rotation planes. To address such cases, Algorithm 2 is adopted across the AMBF Framework, AMBF Simulator, Blender-to-AMBF add-on (III-B) and the URDF-to-AMBF converter (III-A). In the Algorithm, S denotes the Skew-Symmetric matrix, $\vec{a} \times \vec{b}$ denotes the vector cross product, $I_{3\times3}$ is the 3×3 Identity matrix and $\vec{n_x}$, $\vec{n_y}$, $\vec{n_z}$ are the three unit vectors.

Algorithm 2 Convention for Rotation Between Vectors

if $abs(\vec{a}.\vec{b}) \leq 1 - \epsilon$ **then** $R_{\vec{a}}^{\vec{b}} = I_{3\times3}$
else if $\vec{a} \nparallel \vec{b}$ **then**
  $R_{\vec{a}}^{\vec{b}} = I_{3\times3} + S(\vec{a} \times \vec{b}) + S(\vec{a} \times \vec{b})^2(1 - \vec{a}.\vec{b})/(\vec{a} \times \vec{b})^2$
else if $\vec{a} \nparallel \vec{n_x}$ **then** $R_{\vec{a}}^{\vec{b}} = AxisAngle(\vec{a} \times \vec{n_x}, \pi)$
else $R_{\vec{a}}^{\vec{b}} = AxisAngle(\vec{a} \times \vec{n_y}, \pi)$
end if

D. Flexibility of Namespacing and Resource Paths

The foundational structure of AMBF Format allows for the use of multiple namespaces in a single description file.

TABLE I: Simplifying redundant names using namespaces rather than suffixes

URDF & SDF	AMBF Format
/body/limb_⟨left|right⟩	/body/⟨left|right⟩/limb
/box_⟨one|two⟩_lid_⟨top|down⟩	/⟨one|two⟩/box/⟨top|down⟩/lid

This is accomplished by overriding the description file's global namespace with local namespace parameter in the respective body(ies) as shown for **Body B** in Figure 1. Namespacing is not required for joints as their parents and children are searched in all the listed namespaces. This feature of the AMBF not only allows multiple robots and multi-bodies to exist in one description file, but also the ability to create different namespacing for sub-structures of a single robot. One practical example is shown in Table I where the identical bodies are distinguished by namespaces rather than the addition of suffixes to their names. Among other advantages, this allows for the convenience of disseminating distributed controllers using namespaces rather than breaking down link names.

A redundant aspect of URDF or SDF is the specification of resource paths as it is often the case that a robot's visual and collision meshes are located in a single OS directory. However, a qualified path for the mesh needs to be defined for each link. This is somewhat simplified by the use of "package" or "model" tags as base names, which are resolved to the base folder of a "package" or the ".gazebo/model" folder respectively. The AMBF format simplifies this by separating the mesh's name from its path. Towards this end, two global resource paths are defined in the AMBF description's header shown in Figure 1 (for visual and collision geometry). Similar to the global namespace parameter, the mesh resource paths can be overridden locally in the body's description, thus allowing multiple paths in a single description file. Additionally, the mesh path can either be relative or absolute. This greatly improves the readability and manageability of the AMBF description files.

E. Support for Soft Bodies

The AMBF format provides support for soft bodies in addition to rigid bodies. Soft-bodies are defined almost identically to rigid bodies except for additional solver data. The AMBF simulator uses the Bullet's soft body solvers for simulating the interaction. It is recommended to use a significantly lower resolution collision, which is then used as a skeleton for the high density visual mesh. Currently, there are 22 optional parameters that can be used to tune the behavior of a soft body. Some of these parameters include properties such as the magnitude of pressure and volume constraints and elongation, flexion and torsion of underlying nodes (vertices). A detailed discussion of all the parameters is beyond the scope of this manuscript. It should be noted that simulating a soft body does not guarantee real-time dynamic update and has only experimental support in the AMBF simulator at the moment.

TABLE II: Contents of payloads for *afState* and *afCommand* for *afObject* and *afWorld*

afWorld		afObject	
afState	**afCommand**	**afState**	**afCommand**
-	-	Base Frame	Base Frame
Msg Num	-	Msg Num	Msg Num
Server Time	Client Time	Server Time	Client Time
Sim Time	-	Sim Time	-
Num Devs	Ena Throttle	Name	Ena Pos Ctrl
Dyn Freq	Clock	Mass & Inertia	Pose
-	Jump Steps	Transform	Wrench
-	-	Children[]	-
-	-	Joint Names[]	Pos Ctrlr Mask[]
-	-	Joint Positions[]	Joint Cmds[]

F. Communication Payloads

Unlike other robot dynamics simulators, the AMBF simulator does not require any intermediate steps to prepare for bi-directional communication. The bodies defined in the AMBF format are designed to satisfy the *communicability* and *controlability* requirement. An important utility of the AMBF Framework is the capacity to interact with the bodies and constraints using a well-defined communication pipeline. Each body initiates a thread for its bidirectional communication using an Inter Process Communication (IPC) medium (via ROS topics). The outgoing communication provides information about the body's state and is conveniently called the afState message, while the incoming message is called afCommand. Similar to a body, there is a single instance for communication of world's states/commands. The payloads of the World/Body messages are summarized in the Table II.

While most of the payload fields are self-explanatory, some fields are addressed here. In the afState message, the 'children[]' and 'joints[]' fields contain the names of all the connected bodies and children joints to a specific body. The field 'joint positions[]' contains the position of the children joints in order of names in the 'joints[]' field. In afCommand message, the field 'Pose' and 'Wrench' are used to either set the Position or Wrench of the body in 6-DOF space. Switching between controlling the body's position and wrench is achieved using the 'Enable Position Control' field. 'Joint Commands[]' is a command vector to be applied to the children joints of the body in order of names in 'joints[]'. The size of this array does not need to match the size of 'joints[]'. Finally the 'Position Controller Mask[]' field is used to specify which joint commands are position commands versus effort commands and is optional as commands are treated as effort targets by default. The asynchronous nature of AMBF means that all the bodies are capable of running their communication sub-routines in independent threads automatically using the AMBF description file.

III. Compatibility of AMBF with External Software

A. URDF to AMBF Conversion

A significant number of robot models haven already been defined using the URDF format, and arguably, newer robots would continue to be represented using URDF. To take advantage of the existing work and community support for the URDF, a URDF-to-AMBF converter has been developed in parallel with the design of the AMBF format and simulator. The source code of this converter is available at [8]. The converter uses internally implemented XML parsing to reduce the reliance on external ROS parsing packages for portability outside Linux operating systems. As mentioned in the previous sections, URDF is constrained by design to limit the links to a single parent. From a design point of view, this deadlock is enforced by the use of visual and collision offset data in the link description. These offsets are taken from the joint frames of relevant links. For the AMBF format, this visual offset data is used in conjunction with joint data to develop AMBF constraints based on Algorithm 3.

Algorithm 3 URDF Joint to AMBF Joint

1: **if** $JointType := Fixed$ **then** $a\vec{x}_j = \vec{n_z}$ **else** $a\vec{x}_j = joint.Axis$
2: $T^{pv}_j = (T^p_{pv})^{-1} * T^p_j$ ▷ $pv = ParentVisual, p = Parent$
3: $p\vec{v}t_p = P^{pv}_j$, $a\vec{x}_p = R^{pv}_j * a\vec{x}_j$ ▷ $j = Joint, ax = axis$
4: $p\vec{v}t_c = P^{cv}_j$, $a\vec{x}_c = P^{cv}_j * a\vec{x}_j$ ▷ $pvt = Pivot$
5: $ambfR^p_c = RotBetweenVectors(a\vec{x}_c, a\vec{x}_p)$
6: $urdfR^p_c = (R^p_{pv})^{-1} * R^p_j * R^c_{cv}$
7: $R_{jo} = (ambfR^p_c)^{-1} * urdfR^p_c$ ▷ $jo = JointOffset$
8: $a\vec{x}_{jo}, \theta_{jo} = toAxisAngle(R_{jo})$

B. Bi-Directional Usage in Blender

The default simulator for ROS (RViz) does not have the capabilities to generate robot models. The most recent versions of Gazebo provide limited support for generating robot models, but its interface is experimental. Hence, SDF files are often generated using URDF through script converters. URDF files can be created using Solidworks (Solidworks Corp., MA, USA) via **Solidworks2URDF** converter `http://wiki.ros.org/sw_urdf_exporter`. This versatile converter has been in active development and the tool of choice for anyone creating URDFs without handling XML by hand. While ROS and its derivatives are designed to be free for research purposes, Solidworks is not. Not only that, Solidworks lacks support for Linux, which is the OS of choice for ROS related development. It is worth mentioning that the Solidworks2URDF converter lacks bi-directional support in Solidworks (i.e., the generated URDF file cannot be reused to load the corresponding Solidworks assembly). Arguably, this can be attributed to the restrictions posed by Solidwork's plugin API rather than the converter itself.

Even though the AMBF format has a front-end interface to allow for the easy creation of simple robots and mechanisms, a graphical user interface is always helpful in fine-tuning and creating complex robots and multi-bodies. We sought out existing software that can be leveraged for this purpose. A few specifications are outlined for the selection of the corresponding software, which include a "free to share" license, bi-directional API to generate and load models, community support and optionally Linux portability. Based on these specifications, Blender [9] is selected as the graphical interface for creating AMBF description files (Figure 3). Notably, the overall user interface of Blender might offer a relatively steeper learning curve to users unfamiliar with animation software.

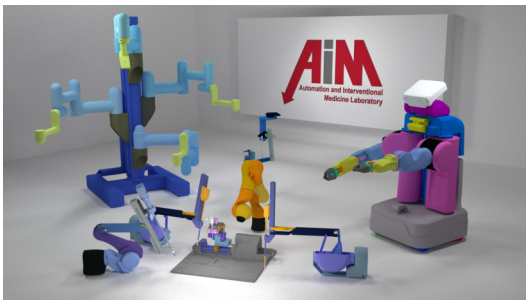


Fig. 3: A subset of robot models already implemented for the AMBF simulator in Blender. These robots include the da Vinci Surgical Robot with multiple parallel mechanisms.

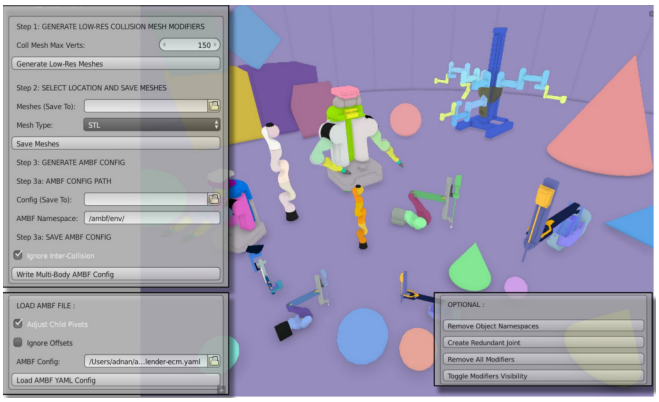


Fig. 4: A few features of the Blender-to-AMBF add-on include copy pasting robot models, scaling, altering the pose of any subset of robots/links, visually setting constraints and inertial properties, creating collision meshes and generating/loading created AMBF description files.

While Blender enjoys huge community support for graphic designers, and hence offers extensive features for such, it has not primarily been used for modeling dexterous robots and bodies with a significant number of interconnected constraints. Blender includes basic support for Bullet Physics for defining constraints and rigid bodies. This support has been leveraged to create a plugin for generating and loading AMBF description files (Figure 4). To include bi-directional usage with Blender, a few simplifications of the AMBF description are required which are addressed in the following subsections. The source code for Blender-to-AMBF add-on is available at [10].

1) Loading AMBFs: Child body's n_z and n_x are the default constraint axis for rotational and translational joints respectively in both Blender and Bullet Physics, while AMBF

format and simulator do not impose this limitation. To enable the same model to be circularly compatible with Blender, the Blender-to-AMBF add-on provides the necessary functionality to alter the multi-body description by adjusting for child body pivots and axis. Adjusting a body axis and pivot is not trivial as all the successor bodies must be accounted for. Since the design philosophy of AMBF separates constraints from bodies, all the body data (meshes) are imported first followed by joints, which in turn connect bodies and enforce world transforms. The pivot and axis correction involves two algorithms which are necessary to make sure that the entire connected structure is bidirectionally compatible:

Algorithm 4 Adjust and Store Child Offsets

if $JointType := Rotational$ **then**
  $a\vec{x}_j = \vec{n_z}$
else $JointType := Translational$
  $a\vec{x}_j = \vec{n_x}$
end if
$R_j^{c^{adj}} \leftarrow RotBetweenVectors(a\vec{x}_j, a\vec{x}_c)$
$T_j^{c^{adj}} := [R_j^c, p\vec{v}t_c]$ ▷ $adj = Adjusted$
$ApplyMeshOffset(T_j^{c^{adj}})$ ▷ $c = Child, pvt = Pivot$
$Body\,Offset\,Map\,[child] \leftarrow T_j^{c^{adj}}$

Figure 5 shows a parent (purple), and a child (turquoise) and the corresponding joint axes marked with the black rings for a rotational joint. To create the constraint, the child's axis is aligned with the parent using Algorithm 2 as shown in Figure 5 (b). As illustrated in the Figure, the child's constraint axis ($\vec{n_y}$) is different from the default axis for rotational constraint type ($\vec{n_z}$). Algorithm 4 is performed iteratively for each constraint before the final Algorithm 6 is performed. The goal here is to offset the body meshes such that the child pivots can be $\vec{0}$ and the child axis is set to the default constraint axis. While performing these mesh offset operations, we keep track of the corresponding imparted offsets so that they can be used later in Algorithm 6. These offsets are stored in a map ($f : body \rightarrow T_j^{c^{adj}}$) where the superscript $T_j^{c^{adj}}$ reiterates that the offset is applied to all bodies when considered as children, however, they are used in Algorithm 6 when treated as parents.

The nature of representing joint data using pivot/axis notation and the correction, thereby using Algorithm 4, can result in axis misalignment along the constraint axis. This misalignment occurs when the rotation due to offset correction occurs outside the plane of rotation between the parent's and child's body axes. The misalignment is best explained by Figure 6. To account for the imparted axis misalignment, Algorithm 5 is performed before Algorithm 6.

After adjusting for the child body offset and axis alignment, the final step is loading all constraints from the AMBF, and consequently, assigning the correct body poses, and eventually, parenting the bodies. This can be summed up with 5 transformations applied iteratively for each constraint to the respective bodies in Algorithm 6.

2) Support for Detatched Joints: As stated in the previous sections, an important goal of the AMBF format is to support

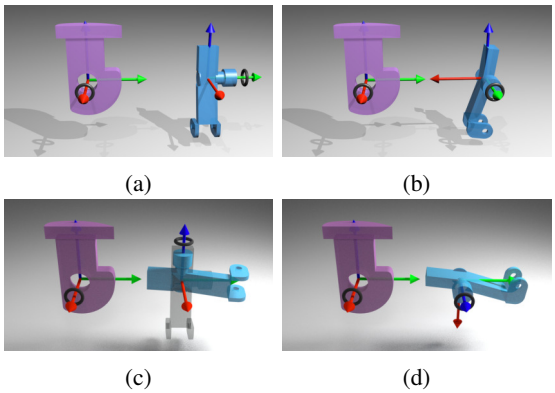


Fig. 5: In the sub-figures, the purple and turquoise bodies represent the parent and child with the constraint axes marked with the black ring. In (b), the child body is rotated to form a constraint by aligning $a\vec{x}_p$ and $a\vec{x}_c$. (c) shows the adjustment required in Blender such that the child body is rotated to adjust the constraint axis to default $\vec{n_z}$ followed by (d) to align the constraint axes with parent's axis.

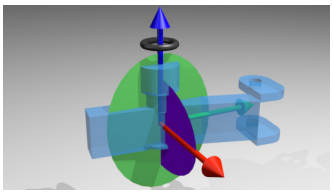


Fig. 6: A visual representation of plane offset between the plane formed by shortest angle rotation between parent's and child's constraint axes (purple disk) and the rotation plane of correction axis (green disk).

closed-loop mechanisms and parallel linkages for robots in an easy manner. While this is conveniently achieved using the front-end syntax of AMBF description format itself, we have provided the necessary means to achieve this using the graphical interface of Blender via Blender-to-AMBF add-on. Blender does not support multiple parents for an object, which is necessary for closed-loop mechanisms. To circumvent this, we use an empty frame with a specific prefix in its name. This empty frame is then used to define a constraint by defining a parent and child body. While parsing through the bodies and constraints in Blender scene to generate an AMBF description file, we leverage the

Algorithm 5 Axis Alignment

$R_j^p = RotBetweenVectors(a\vec{x}_j, a\vec{x}_p)$
$R_c^{p^{adj}} = R_j^p R_c^{j^{adj}}$
$R_c^p = RotBetweenVectors(a\vec{x}_c, a\vec{x}_p)$
$R_{off} = (R_c^{p^{adj}})^{-1} * R_c^p$
$a\vec{x}_{off}, \theta_{off} = ToAxisAngle(R_{off})$
if $\theta_{off} > \epsilon$ **then**
  $R_{ao} = FromAxisAngle(a\vec{x}_c, \theta_{off})$
  $Body\,Offset\,Map\,[child] \leftarrow R_{ao} T_j^{c^{adj}}$
else
  $Body\,Offset\,Map\,[child] \leftarrow I_{4\times4}$
end if

Algorithm 6 Pose Data from AMBF Format

1: **if** $JointType := Rotational$ **then**
2: $\quad a\vec{x}_j = \vec{n_z}$
3: **else**$JointType := Translational$
4: $\quad a\vec{x}_j = \vec{n_x}$
5: **end if**
6: $T_p^w \leftarrow$ Parent Body's Pose in World
7: $T_{bo} \leftarrow Body\,Offset\,Map\,[parent]$
8: $T_j^p = [I_{3\times3}, \vec{pvt_p}]$
9: $T_{jo} = [R_{jo}, 0]$ ▷ $R_{jo} = FromAxisAngle(a\vec{x}_j, \theta_{jo})$
10: $T_c^j = [R_c^j, 0]$ ▷ $R_c^j = RotBetweenVectors(a\vec{x}_c, a\vec{x}_j)$
11: $T_c^{\hat{w}} = T_p^{\hat{w}}\,T_{bo}\,T_j^p\,T_{jo}\,T_c^j$

assigned naming prefix to treat the empty frame as a place holder rather than an actual empty body. The allows us to create densely connected bodies robustly without having to manually touch up the AMBF description file.

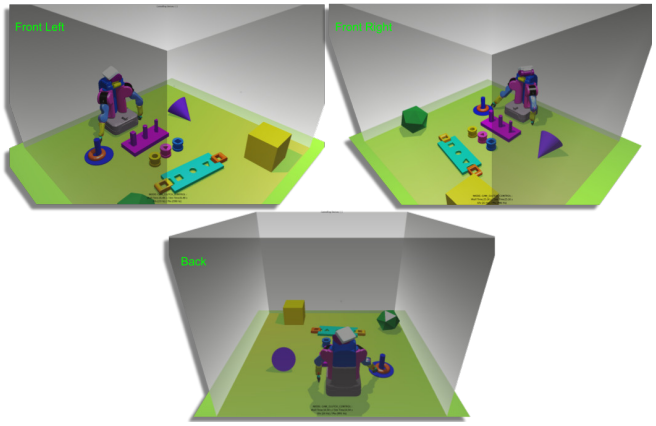


Fig. 7: A multi-port view of the underlying simulation using 3 frame-buffers which output to separate windows and can be dragged around different monitors.

C. Implementation of Multiple View-ports using Camera Data

One of the design requirements of the AMBF framework and the AMBF simulator is the ability to manipulate multi-manual tasks in the real-time dynamic simulation with haptic feedback. The design goal enables multiple users alongside AI to share a simulation via haptic/input devices. To this end, having multi-port frame buffers can assist the users in performing tasks with their own perspective viewports. Additionally, the users should potentially posses the camera control for their view-port independent of the other users in the simulation. As a result, the prospective design of the AMBF description format includes support for achieving multiple viewports and binding input devices to each (described in Figure 1 for the camera tile). Figure 7 shows the result of using multiple cameras and thereby achieving multiple views/windows of the underlying simulation.

IV. Results and Discussion

The PC setup used for the results in this section consists of an Intel(R) Core(TM) i7-3770 CPU (3.40GHz), Fujitsu 32 GB DDR3 RAM (1333 MHz) and an Nvidia GTX 1060 (8

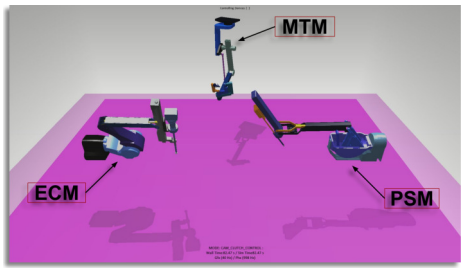


Fig. 8: A simulation with several manipulators running in real-time. The labeled manipulators (ECM and PSM) have two connected closed-loop mechanism while the MTM has one closed-loop mechanism.

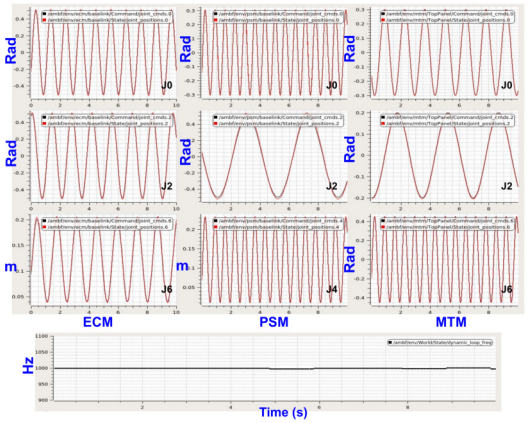


Fig. 9: Each column shows the joint control of a different manipulator labeled underneath. The last row shows the dynamic update frequency of physics simulation. The ECM's and PSM's 3rd graph depicts a translational joint while all the joints of the MTM are rotational.

GB RAM) GPU running Ubuntu 18.04. We demonstrate the controller performance of multiple closed-loop robots in the simulation environment using ROS communication as IPC. The robots shown in Figure 8 are commanded at 1 kHz. The joints are controlled in position control mode and labeled in Figure 9. The inertial parameters of the PSM and MTM have been computed by Yan et.al [11] and can be utilized in the AMBF description files. As shown in Figure 9, the joint response at 1 kHz of communication frequency remains stable and robust.

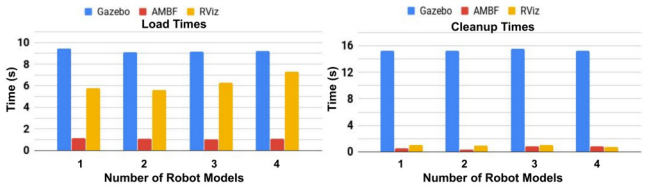


Fig. 10: The loading vs unloading times for simulators with increasing number of complex robot models. The simulators are loaded using the bash terminal.

The AMBF simulator was designed to reduce computational overheads and enable efficient loading and unloading

of models. This also assists in the workflow of developing a multi-body representation using Blender-to-AMBF add-on and quickly loading it in the AMBF Simulator. We present the loading times of multiple robots in parallel with multiple closed loop constraints and compare them with Gazebo and RViz using similar models and identical system load. As evident from Figure 10, not only does the AMBF simulator outperform Gazebo and RViz in terms of loading speed and controller performance, it also outperforms in the cleanup speed.

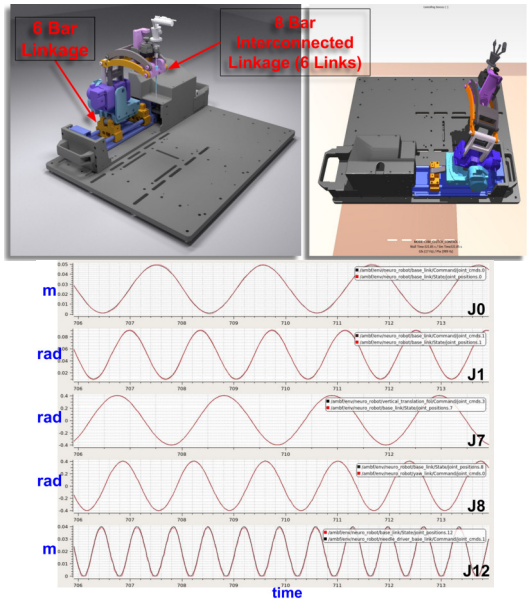


Fig. 11: WPI's Neuro Surgery Robot Model using the Blender-to-AMBF add-on. The robot consists of a 6 bar linkage at the base and an inter-connected 8 bar linkage at the top. The robot is controlled using ROS topics at 1 kHz communication frequency.

Another demonstration of a complex robot (WPI's Neuro Robot [12]) and its controller performance is shown is Figure 11. The URDF description of the robot was developed at [13] and is converted using the URDF-to-AMBF converter. The AMBF model is then loaded in Blender using the Blender-to-AMBF add-on to create parallel linkages and then adding visual details and colors to the Robot. Using the IPC controllers, various joints of the robot have been excited to follow a different sinusoidal frequency.

Finally, we present a key feature of the AMBF Simulator (and the AMBF format) where several input devices (haptic/tracker) can be used to interact with a dynamic environment shown in Figure 12. The goal here is to demonstrate the application of real-time dynamic haptic interaction with simulated robots to perform tasks with shared autonomy. In addition to interacting with simulated robots, the input devices can be bound to any camera (view-ports shown in Figure 7) which allow for hand-eye coordination w.r.t. the camera and also allow the devices to move the camera itself. A supplementary video demonstrates the additional features of the AMBF Simulator.

Future goals for the AMBF Simulator and Framework include bi-directional support with URDF and SDF and improved performance of soft body simulations. Dedicated Graphics Processing Units (GPU) need to be leveraged to improve the speed of Softbody simulations. Some work has already been done, but the nature of GPU compute languages and varying features of different GPUs makes portability difficult. The source code for the AMBF Framework, Simulator and its supported software can be found at [14].

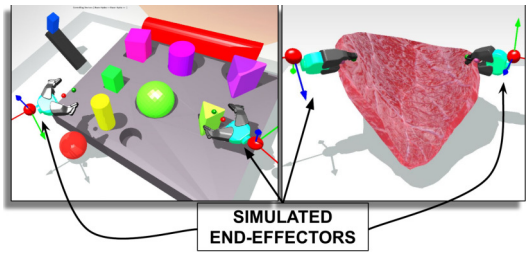


Fig. 12: Interaction with Simulated Multi Manual Puzzles defined using AMBF Format.

References

[1] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, Sep. 2004, pp. 2149–2154 vol.3.

[2] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012, pp. 5026–5033.

[3] E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov 2013, pp. 1321–1326.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[5] A. Agrawal and R. Gondokaryono, "Accurate URDF and SDF models of Intuitive Surgical's daVinci Research Kit (dVRK)," https://github.com/charlespwd/project-title, 2018.

[6] E. Coumans, "Bullet Physics Simulation," in *ACM SIGGRAPH 2015 Courses*, ser. SIGGRAPH '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2776880.2792704

[7] C. et al., "The CHAI libraries," in *Proceedings of Eurohaptics 2003*, Dublin, Ireland, 2003, pp. 496–500.

[8] A. Munawar, "A versatile Universal Robot Description Format (URDF) to Asynchronous Multi-Body Framework (AMBF) Format Converter," https://github.com/WPI-AIM/urdf_2_ambf, 2019.

[9] R. Hess, *Blender Foundations: The Essential Guide to Learning Blender 2.6*. Focal Press, 2010.

[10] A. Munawar, "A Graphical add-on for Blender for Creating and Loading AMBF Models," https://github.com/WPI-AIM/ambf_addon, 2019.

[11] Y. Wang, R. Gondokaryono, A. Munawar, and G. S. Fischer, "A Convex Optimization-based Dynamic Model Identification Package for the da Vinci Research Kit," *IEEE Robotics and Automation Letters*, pp. 1–3, 2019.

[12] C. J. Nycz, R. Gondokaryono, P. Carvalho, N. Patel, M. Wartenberg, J. G. Pilitsis, and G. S. Fischer, "Mechanical validation of an MRI compatible stereotactic neurosurgery robot in preparation for pre-clinical trials," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2017, pp. 1677–1684.

[13] C. Bove, "Constrained motion planning system for mri-guided, needle-based, robotic interventions," Master's thesis, Worcester Polytechnic Institute, 2018.

[14] A. Munawar, "The Asynchronous Multi-Body Framework," https://github.com/WPI-AIM/ambf, 2019.