Rigid body pose and twist scene graph founded on geometric relations semantics for robotic applications

Tinne De Laet, Herman Bruyninckx, and Joris De Schutter

Abstract— **This paper presents a scene graph for geometric relations between rigid bodies that keeps track of poses and twists of rigid bodies in the scene. The scene graph relies on semantic pose and twist representation, making it invariant to the actual coordinate representation at hand. This makes the scene graph more general and interoperable than most scene graphs currently available. The presented scene graph takes into account constraints imposed by particular coordinate representations, allows for constant poses, answers semantic pose and twist queries, and provides built-in semantic consistency checks. Since the scene graph also keeps track of the twist, it allows native twist calculations, as opposed to deriving the velocities from the poses in the graph. This paper comes with software released under a dual BSD/LGPLv2.1 license.**

I. Introduction

In robotics, questions like "Where is the gripper of my robot with respect to the door handle?" or "What is the speed of my head camera with respect to the tracked object?" are very typical. These questions arise in different application scenarios such as robot programming, robot navigation, sensor processing,...To provide answers, time-dependent geometric relationships such as relative position, orientation, pose (position and orientation), linear velocity, angular velocity, and twists (linear and angular velocities) between objects in the scene have to be calculated. To this end the robot programmer can build a **scene graph**, keeping track of the geometric relationships between the rigid bodies in the scene.

The name "scene graph" is often used for graphs that arrange the logical and/or spatial representation of a scene. Scene graphs are used in a wide range of applications and domains: vector-based graphics editing applications, computer games, 3D creation suites like Blender, but also in robotics. This has led to a proliferation of different scene graph implementations, based on the same principles but adapted to the particular application or domain at hand. This paper focuses on scene graphs that model spatial representations in a robotics scene. Such graphs, mostly modelling the orientation and position between different objects in the scene, are also referred to as "transform graphs".

All authors are with the Department of Mechanical Engineering, KU Leuven, Belgium. Corresponding author: Tinne De Laet (`Tinne.DeLaet@mech.kuleuven.be`)

Via the scene graph, the robot programmer can automatically retrieve time-dependent geometric relationships between objects in the scene. However, to express and calculate geometric relations and perform mathematical operations on them (e.g. composition of relative motion, time differentiation, or integration), robot programmers have to choose coordinate representations with which to perform the corresponding numerical operations. Despite a history of over 50 years, the geometric properties of rigid-body operations, and their coordinate representations, have never been standardized. All exisiting representations entail a surprisingly large number of choices or assumptions, which are often made implicitly. Not explicitly stating these assumptions may lead to errors in the calculations (composition of geometric relations expressed in different coordinate frames, composition of poses and orientation coordinate representations in wrong order, ...) [1]. To alleviate this problem, we recently proposed semantics for the standardization of geometric relations between rigid bodies [1], [2], referred to as 'geometric semantics'. These semantics explicitly state the coordinate-invariant properties and operations, and, more importantly, all the choices that are made in coordinate representations of these geometric relations. This results in a set of concrete suggestions for standardizing terminology and notation, allowing programmers to write fully unambiguous software interfaces, including automatic checks for semantic correctness of all geometric operations on rigid-body coordinate representations. Furthermore, a C++ implementation is available as open-source [3], [4].

The goal of this paper is to show how the geometric semantics can be used as primitives for scene graphs with built-in semantic consistency checks. In particular, this paper develops a *scene graph* that keeps track of rigid body *poses* and *twists*. We will show how the development of the scene graph profits from the geometric semantics. This paper advances the state of the art in robotics in different ways: 1) it presents the first scene graph explicitly supporting and relying on the *semantics for geometric relationships between rigid bodies*; 2) to our knowledge it is the first scene graph that keeps track of the *twists*, hereby allowing native calculations with twists; 3) it allows for online *semantic queries* on the scene graph, 4) it naturally accommodates for *constant poses*, 5) it can use *any par-*

ticular coordinate representation, and 6) it automatically *checks the constraints imposed by the particular coordinate representations*. Finally, this paper comes with pointers to all software released under a dual BSD/LGPLv2.1 license.

Section II gives an overview of related work. Section III provides a short summary of the geometric semantics theory relevant for this paper. Section IV explains the design, implementation, and the use of the scene graph. Section V contains a robotics example illustrating the use of the scene graph and the accompanying software. Finally, Section VI discusses the contributions of this paper and points to future work.

II. RELATED WORK

This paper does not aim at giving an overview of the entire scene graph research (for this we refer to the history overview of scene graphs [5]), but focuses on scene graphs that model spatial representations in a robotics "world model".

The Robotics Library [6] provides a scene graph that is built from a collection of models (specified in the VRML format), where each model consists of several bodies with shapes. The scene graph supports the use of kinematic models (chains) and allows to calculate collisions between the objects in the scene.

Another project worth to mentioning is the OpenScene-Graph [7] as it is a widely used high performance 3D graphics toolkit in fields such as visual simulation, games, virtual reality, scientific visualization and modeling. As for most scene graphs originating from the 3D graphics community, this project rather targets to the 3D graphical representation rather than to the underlying transform graph.

The most used transform graph in robotics is tf [8], or its second generation descendant tf2 [9], [10]. tf and tf2 keep track of multiple coordinate frames over time, maintain the relationship between coordinate frames in a tree structure buffered in time, and let the user transform the coordinates of points, vectors, ... between any two coordinate frames at any desired point in time. The core tf2 package is a ROS-independent implementation, with additional tools to use tf2 within ROS. Recently, tf has been extended to represent uncertainty in the pose while however keeping the underlying tf tree structure [11].

III. GEOMETRIC SEMANTICS, BACKGROUND [1]

Geometric relationships between bodies are described using a set of *geometric primitives*[1]: points (e), vectors, orientation frames ([a] representing an orientation by means of three orthonormal vectors indicating the frame's X,

[1]This background contains a short summary of the semantics for the standardization of geometric relations between rigid bodies, for more details we refer to [1].
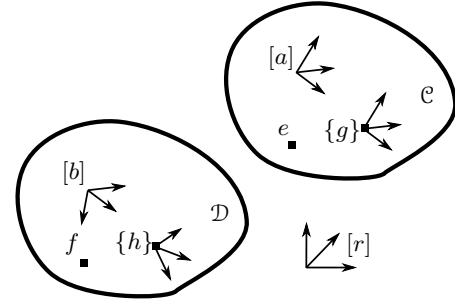

Fig. 1: The geometric primitives that are useful to define geometric relationships of body $\mathcal{C}$ with respect to body $\mathcal{D}$ are: an orientation frame [a], a point e, and frame $\{g\}$ fixed to body $\mathcal{C}$, an orientation frame [b], a point f, and frame $\{h\}$ fixed to body $\mathcal{D}$, and a coordinate frame [r], considered instantaneously fixed to body $\mathcal{D}$, in which the coordinates are expressed. (Extract from [1].)

Semantics	Geometric primitives
PoseCoord $((e, [a]) \mid \mathcal{C}, (f, [b]) \mid \mathcal{D}, [r])$	point e
	orientation frame $[a]$
	body $\mathcal{C}$
	reference point f
	reference orientation frame $[b]$
	reference body $\mathcal{D}$
	coordinate frame $[r]$
PoseCoord $(\{g\} \mid \mathcal{C}, \{h\} \mid \mathcal{D}, [r])$	frame $\{g\}$
	body $\mathcal{C}$
	frame $\{h\}$
	reference body $\mathcal{D}$
	coordinate frame $[r]$
TwistCoord $(e \mid \mathcal{C}, \mathcal{D}, [r])$	point e
	body $\mathcal{C}$
	reference body $\mathcal{D}$
	coordinate frame $[r]$

TABLE I: Minimal coordinate semantics for the pose and twist of body $\mathcal{C}$ with point e, orientation frame [a], and frame $\{g\}$ with respect to $\mathcal{D}$ with point f, orientation frame [b], and frame $\{h\}$ with coordinates expressed in coordinate frame [r]. (Extract from [1])

Y, and Z-axes), and frames ($\{g\}$). Figure 1 graphically presents the geometric primitives body, point, vector, orientation frame, and frame. We will consistently use the following naming for the geometric primitives to represent the geometric relationship of a body $\mathcal{C}$ with respect to body $\mathcal{D}$ in this document: $e \mid \mathcal{C}$, $[a] \mid \mathcal{C}$, $\{g\} \mid \mathcal{C}$, $f \mid \mathcal{D}$, $[b] \mid \mathcal{D}$, and $\{h\} \mid \mathcal{D}$.

Table I summarizes the minimal but complete set of geometric primitives and the semantics for the geometric relations pose and twist between rigid bodies, which are the most relevant relationships for this paper.

IV. Scene graph

Before discussing the proposed design of the scene graph it is important to list the **envisaged goals**: 1) The scene graph is a *physically correct and consistent representation* of the poses and twists of the rigid bodies in the scene, 2) The scene graph answers *semantic queries*[2] such as PoseCoord $((e, [a])\,|\mathcal{C}, (f, [b])\,|\mathcal{D}, [r])$ and TwistCoord $(e|\mathcal{C}, \mathcal{D}, [r])$ *online*, 3) Instead of enforcing a particular coordinate representation or to silently convert the user's coordinate representation to the scene graph's internal coordinate representation, the proposed scene graph uses the coordinate representation chosen by the user and uses this coordinate representation internally. 4) The scene graph handles the constraints imposed by the particular coordinate representation. 5) Finally, the scene graph provides a solid but flexible basis for future extensions (for instance including uncertainties).

The following **general design and development decisions** were made: 1) A scene graph combining the pose and twist relationships is made. Most scene graph implementations use the pose as native type and derive the twists from the available poses by means of finite differences, which is not optimal for all use cases and applications. Imagine the case where the velocity of a rigid body is measured. If only poses are kept in the scene graph, the twist of the body has to be integrated to a pose, which is subsequently added to the scene graph. If afterwards, a query is made for this twist, the pose has to be differentiated again to obtain the twist that was actually measured. These conversions are avoided when adding the twist relationships in the scene graph. 2) The geometric semantics [1] are used as a basis for the scene graph since: a) They provide the minimal but complete semantics of the poses and twists and hereby *facilitate interoperability and reduce application and, especially, system integration development time*; b) The scene graphs can *check the physical correctness and consistency* of the scene representation; c) They provide a *semantic wrapper for geometric operations* (composition, integration,...) so abstraction can be made of the particular coordinate representation operations (multiplication, summation,...), which allows to postpone the decision on the particular coordinate representation *until* the user has decided which coordinate representation suits his needs best. d) After this choice, the geometric semantics allows to *model and check the constraints imposed by that particular coordinate representation*. 3) Rather than building yet another graph library next to all the existing ones [12]–[14], the proposed scene graph can use existing graph libraries (in this case

the Boost Graph Library [12]) since a) it already provides *an efficient implementation of graph operations and allows to traverse the graph efficiently*, and b) the general graph structure will *accommodate future extensions* (e.g. when including uncertainties, in which case the scene graph will no longer be a directed forest (see Sections IV-B and IV-C)).

The scientific challenge of the scene graph is to construct a graph that complies with all the constraints that exist between poses themselves, between twists themselves, and between poses and twists. Some examples: 1) PoseCoord $(\{a\}|\mathcal{A}, \{c\}|\mathcal{C}, [c])$ should be equal to the composition of PoseCoord $(\{a\}|\mathcal{A}, \{b\}|\mathcal{B}, [b])$ and PoseCoord $(\{b\}|\mathcal{B}, \{c\}|\mathcal{C}, [c])$, hereby introducing a constraint between these three poses; 2) TwistCoord $(a|\mathcal{A}, \mathcal{C}, [a])$ should be equal to the composition of TwistCoord $(a|\mathcal{A}, \mathcal{B}, [a])$ and TwistCoord $(a|\mathcal{B}, \mathcal{C}, [a])$, hereby introducing a constraint between these three twists; 3) TwistCoord $(a2|\mathcal{A}, \mathcal{C}, [a2])$ should be equal to the result of changing the point and coordinate frame of TwistCoord $(a|\mathcal{A}, \mathcal{C}, [a])$ using PoseCoord $(\{a\}|\mathcal{A}, \{a2\}|\mathcal{A}, [a2])$, hereby introducing a constraint between these two twists and the pose. A scientific contributions of this paper is the identification of all the constraints the scene graph should satisfy such that it is a *minimal, physically correct, and consistent representation* of the scene (i.e. all the constraints that exist between poses and twists).

The next sections will discuss the scene graph's nodes, representing the rigid bodies and the frames, and the edges, representing the pose and twist relationships.

A. Rigid bodies and frames

The scene graph contains information about relative poses and twists between rigid bodies. These poses and twists define *relationships* between frames, which are fixed to rigid bodies. Therefore the **nodes** in the scene graph will represent **frames and rigid bodies**, while the edges will represent the **pose and twist** relationships between the frames and rigid bodies.

This leads to the following **node** definitions: 1) a **rigid body node** represents a rigid body and contains all the frames fixed to this rigid body; and 2) a **frame node** contains a frame and is contained in the corresponding rigid body node. The nodes in the scene graph have to satisfy the following **constraints**: 1) *Unique rigid bodies:* Since a rigid body refers to a unique physical entity, each rigid body can only occur once, 2) *Unique frames:* Since a frame can physically only be attached to a single rigid body and it represents a unique physical entity, each frame can only occur once. 3) *Frame attached to rigid body:* Since a frame is always physically attached to a rigid body, each frame node should be contained in the rigid body

---

[2]The query is called "semantic" because it looks for the particular coordinate representation (i.e. the actual numbers) corresponding to a particular geometric relationship defined using the geometric semantics [1].
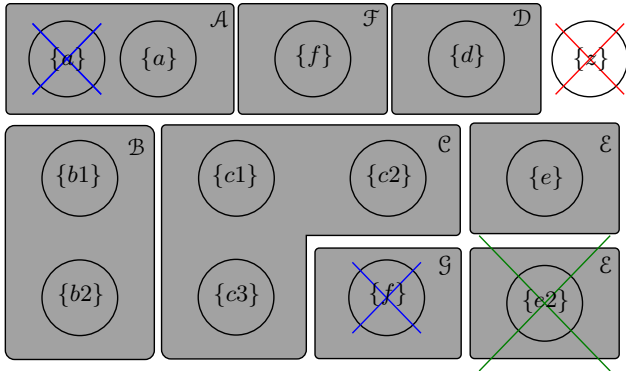
Fig. 2: A scene graph has two types of nodes: rigid body nodes and frame nodes. They have to satisfy the following **constraints**: 1) *Unique rigid bodies:* Adding rigid body $\mathcal{E}$ is not allowed since it already exists. 2) *Unique frames:* Adding frames $\{a\}$ and $\{f\}$ is not allowed since they already exist. 3) *Frame attached to rigid body:* Adding $\{z\}$ is not allowed since it is not mentioned to which rigid body it is attached.

node representing the rigid body it is physically attaced to. Figure 2 illustrates the node definitions and constraints for the scene graph.

B. Pose relationships

Since the scene graph contains relative poses between rigid bodies, which entails a directed relationship, the pose relationships are **directed edges called "pose edges"**. We impose that the **poses are expressed in the reference orientation frame**, i.e. the coordinate frame equals the reference orientation frame, and that the **poses are between two frames**, i.e. the (reference) point and the (reference) orientation frame correspond to the origin and the orientation of the frame on the (reference) body, respectively. This is inspired by the facts that: 1) poses are most often expressed using full frames instead of a separate point and orientation frame [1], and 2) composing poses is of multiplicative nature [1]. This leads to the following definition: A **pose edge** in the scene graph contains the coordinates of the relative pose of the frame in the head node with respect to the frame in the tail node and expressed in the frame of the tail node. This definition entails the following constraint on the scene graph: 1) *Pose edges:* A pose edge is always defined between two <u>frame</u> nodes. Figure 3 illustrates the pose edge definition and the constraint imposed by this definition.

The physical correctness and consistency of the pose relationships are guaranteed by checking two constraints: 1) *Constant pose:* The pose between two frames both fixed to the same rigid body is constant, therefore it is not allowed to change this pose once it is set; 2) *Pose*
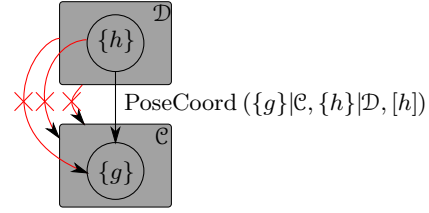

Fig. 3: The pose edge contains the coordinates of the relative pose of the frame in the head node with respect to the frame in the tail node and expressed in the frame of the tail node, therefore the pose edges should always be defined between two frame nodes. The crossed pose edges illustrate poses not satisfying this constraint.

is uniquely defined: The pose between two rigid bodies should be uniquely defined leading to two constraints: a) The pose of a frame should only defined with respect to a **single reference frame**. Therefore each node can only have one parent, making the pose graph a forest. b) There should exist **maximally one pose path between two pairs of rigid bodies**. Therefore, when adding a pose edge no descendant of the head is allowed to have the same body as any descendant of the root of the source. This makes the pose relationships in the scene graph a constrained graph. The above constraints make the resulting pose relationships in the scene graph a **constrained directed forest**. Figure 4 graphically illustrates the constraints.

C. Twist relationships

Since the scene graph contains the information of relative twists between rigid bodies, which entails a directed relationship, the twist relationships are **directed edges, called twist edges**. For the twist edges we impose that the **twists are screw twists**, i.e. the point of the twist is the origin of the frame on the body and the coordinate frame of the twist is the orientation frame of the frame on the body. This is inspired by the facts that: 1) the minimal coordinate semantics of a twist consists of a point e, often referred to as the velocity reference point, and a coordinate frame $[r]$ (Section III), and 2) only a <u>screw</u> (or body-fixed) twist can be integrated to a pose and the time derivative of a pose relationship is a screw twist [1]. Therefore, using screw twist allows us to easily connect the pose changes over time with the twists. This leads to the following definition: A **twist edge** in the scene graph contains the coordinates of the relative screw twist of the body of the frame in the head frame node with respect to the body in the tail rigid body node, i.e. with velocity reference point corresponding to the origin of the frame in the head frame node and with coordinate frame the orientation frame of the frame in the head frame node. This definition entails the following constraint on the scene graph: 1) *Twist edges:* The twist
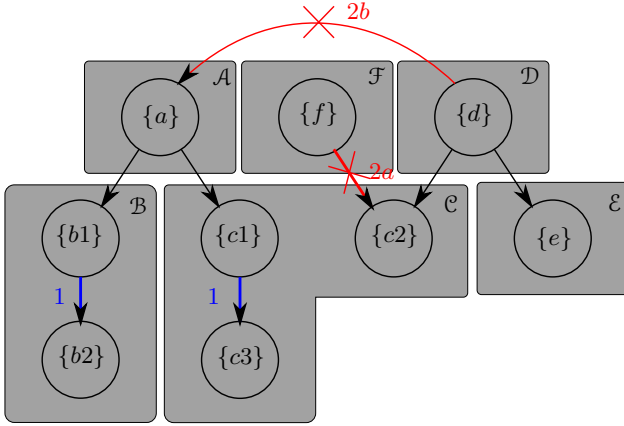
Fig. 4: The figure illustrates the constraints on a scene graph needed to guarantee physical correctness and consistency of the pose relationships: 1) *Constant pose:* The pose of $\{b1\}$ with respect to $\{b2\}$ is constant since the frames are both fixed to rigid body $\mathcal{B}$; 2) *Pose is uniquely defined:* a) Adding an edge from $\{f\}|\mathcal{F}$ to $\{c2\}|\mathcal{C}$ (i.e. adding PoseCoord$(\{c2\}|\mathcal{C}, \{f\}|\mathcal{F}, [f])$) is not allowed since the pose of $\{c2\}|\mathcal{C}$ is already defined with respect to $\{d\}|\mathcal{D}$ (i.e. by PoseCoord$(\{c2\}|\mathcal{C}, \{d\}|\mathcal{D}, [d])$); b) Adding an edge from $\{d\}|\mathcal{D}$ to $\{a\}|\mathcal{A}$ (i.e. adding PoseCoord$(\{a\}|\mathcal{A}, \{d\}|\mathcal{D}, [d])$) is not allowed since this would result in two possible ways to determine the pose of $\mathcal{C}$ with respect to $\mathcal{D}$, i.e by PoseCoord$(\{c1\}|\mathcal{C}, \{d\}|\mathcal{D}, [d])$ or by PoseCoord$(\{c2\}|\mathcal{C}, \{d\}|\mathcal{D}, [d])$.

edges are always defined between a rigid body node and a frame node. Figure 5 illustrates the twist edge definition and the constraint imposed by this definition.

The physical correctness and consistency of the pose relationships are guaranteed by checking one constraint: 1) *Twist is uniquely defined:* The twist between two rigid bodies should be uniquely defined leading to the constraints: a) The twist of a frame on a rigid body should only be defined with respect to a **single reference body**. Therefore each frame node can only have one parent through a twist relationship. This makes the twist relationships in the scene graph a forest. b) There should exist **maximally one twist path between a rigid body and a set of frames on a rigid body that are pose connected**. Therefore, when adding a twist edge no other frame on the rigid body of the head that is connected through a pose path with the head frame node is allowed to have a twist path to the tail rigid body node. Otherwise the twist information to be added is already contained in the scene graph: the twist can be calculated using the poses and twists already present. This makes the twist relationships in the scene graph a constrained graph. The above constraints make the resulting twist relationships in
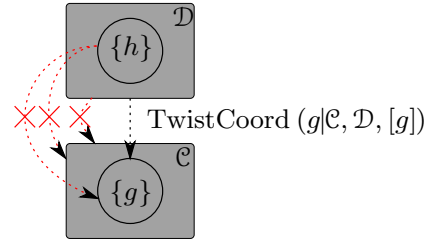

Fig. 5: The twist edges contains the coordinates of the relative screw twist of the frame in the head frame node with respect to the rigid body in the tail rigid body node, therefore the twist edges are always defined between a rigid body node and a frame node. The crossed twist edges illustrate twists not satisfying this constraint.
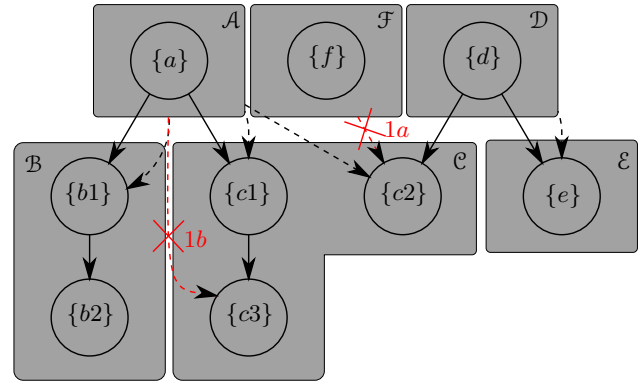

Fig. 6: The figure illustrates the constraints on a scene graph needed to guarantee physical correctness and consistency of the twist relationships: 1) *Twist is uniquely defined:* a) Adding a twist edge from $\mathcal{F}$ to $\{c2\}|\mathcal{C}$ (i.e. adding TwistCoord$(c2|\mathcal{C}, \mathcal{F}, [c2])$) is not allowed since the twist of $\{c2\}|\mathcal{C}$ is already defined with respect to $\mathcal{D}$ (i.e. by TwistCoord$(c2|\mathcal{C}, \mathcal{D}, [c2])$). b) Adding a twist edge from $\mathcal{A}$ to $\{c3\}|\mathcal{C}$ (i.e. adding TwistCoord$(c3|\mathcal{C}, \mathcal{A}, [c3])$) is not allowed since the twist of $\{c3\}|\mathcal{C}$ is already defined with respect to $\mathcal{A}$ (i.e. by TwistCoord$(c1|\mathcal{C}, \mathcal{A}, [c1])$ and PoseCoord$(\{c3\}|\mathcal{C}, \{c1\}|\mathcal{C}, [c1])$). Pose and twist relationships are indicated with solid and dashed arrows, respectively.

the scene graph a **constrained directed forest**. Figure 6 graphically illustrates the constraints.

D. Queries

The purpose of the proposed design for the scene graph was to answer questions on the geometric relations between rigid objects in the scene. This section details the supported semantic queries and the operations needed to answer them.

Possible semantic queries are the relative position PositionCoord$(e|\mathcal{C}, f|\mathcal{D}, [r])$, orientation OrientationCoord$([a]|\mathcal{C}, [b]|\mathcal{D}, [r])$, pose

PoseCoord $((e, [a]) | \mathcal{C}, (f, [b]) | \mathcal{D}, [r])$, linear velocity LinearVelocityCoord $(e | \mathcal{C}, \mathcal{D}, [r])$, angular velocity AngularVelocityCoord $(\mathcal{C}, \mathcal{D}, [r])$, or twist TwistCoord $(\{e\} | \mathcal{C}, \mathcal{D}, [r])$ between rigid objects in the scene. As an example Algorithms 1-3 provide (highly simplified) pseudocode for the semantic query GetPose(PoseCoord $((e, [a]) | \mathcal{C}, (f, [b]) | \mathcal{D}, [r]))$. Three

---

Algorithm 1: GetPose(PoseCoord $((e, [a]) | \mathcal{C}, (f, [b]) | \mathcal{D}, [r])$) pseudocode

---

   Input : semantics of pose to look for
   Output: the coordinates of the found pose

1   result = GetBasicPose(PoseCoord $(\{a\} | \mathcal{C}, \{b\} | \mathcal{D}, [b])$);
   // result = PoseCoord $(\{a\} | \mathcal{C}, \{b\} | \mathcal{D}, [b])$
2   **if** $f \mathrel{!=} b$ **then**   // ref point != ref orientation frame
3   pos = GetPosition(PositionCoord $(f | \mathcal{D}, b | \mathcal{D}, [b])$);
4   result.changeRefPoint(pos.inverse());
   // result = PoseCoord $(\{a\} | \mathcal{C}, (f, [b]) | \mathcal{D}, [b])$
5   **if** $e \mathrel{!=} a$ **then**   // point != orientation frame
6   pos = GetPosition(PositionCoord $(e | \mathcal{C}, a | \mathcal{C}, [b])$);
7   result.changePoint(pos);
   // result = PoseCoord $((e, [a]) | \mathcal{C}, (f, [b]) | \mathcal{D}, [b])$
8   **if** $r \mathrel{!=} b$ **then**   // coordinate frame != ref orient frame
9   or=GetOrientation(OrientationCoord $([b] | \mathcal{D}, [r] | \mathcal{R}, [r])$);
10   result.changeCoordinateFrame(or);
   // result = PoseCoord $((e, [a]) | \mathcal{C}, (f, [b]) | \mathcal{D}, [r])$
11   **return** result;

Algorithm 2: GetBasicPose(PoseCoord $(\{a\} | \mathcal{C}, \{b\} | \mathcal{D}, [b])$) pseudocode

---

   Input : semantics of pose to look for
   Output: the coordinates of the found pose

1   ancestor =FindYoungestCommonAncestor($\{a\}, \{b\}$);
2   pathTarget =FindPath(ancestor,$\{a\}$);
3   pathSource =FindPath(ancestor,$\{b\}$);
4   poseTargetAncestor=ComposePosesPath(pathTarget);
5   poseSourceAncestor=ComposePosesPath(pathSource);
6   result =compose(poseSourceAncestor.inverse(), poseTargetAncestor);
7   **return** result;

---

types of operations occur: 1) graph operations, which boil down to operations on the constrained forest using the boost graph library, 2) semantic geometric operations, provided by the geometric semantics, which provide a semantic wrapper for the actual geometric calculations, and 3) composite operations, involving both graph operations and semantic geometric operations. These operations raise errors in case the operations are not

---

successful. In particular the semantic geometric operations raises errors if the constraints imposed by the particular coordinate representation are not met. For example: if the homogeneous transformation matrix is chosen as a pose coordinate representation, it imposes semantic constraints (the origin and the orientation of the same frame on the body and reference body are used ((reference) point = origin of (reference) orientation frame), and the relative pose is expressed in the orientation frame fixed to the reference body (reference orientation frame = coordinate frame). In this example therefore, any query not satisfying these constraints will be unsuccessful. For example: semantic geometric operations such as changeRefPoint (Algorithm 1, line 4), changePoint (Algorithm 1, line 7), and changeCoordinateFrame (Algorithm 1, line 10) will fail.

Algorithm 3: ComposePosesPath(path) pseudocode

   Input : path along which to compose poses
   Output: pose of path head node wrt path tail node

1   **for** *edge* = *path.begin()* **to** *path.end()* **do**
2   result = compose(result,edge.pose);
3   **return** result;

E. Implementation

The scene graphs C++ implementation [15] is publicly available under a dual BSD/LGPLv2.1 license. It is built on top of the geometric semantics library [4] and the Boost Graphical Library [12]. The core software is templated, such that the user can still select the most suited coordinate representation for the application at hand, an implementation using the KDL [16] coordinate representations is already available.

We want to emphasize how the development and implementation process was facilitated and accelerated thanks to the use of the geometric semantics. Since the geometric semantics provide the minimal but complete semantic information, they were very helpful when designing the different functions by explicitly writing down what every function is semantically changing. Furthermore, during the implementation process, the geometric semantics were continuously providing semantic checking of all the implemented functions. This way errors could be detected early in the development process.

V. EXAMPLE

The purpose of the scene graph is to answer queries on the relative position, orientation, pose, linear velocity, angular velocity, or twist of different objects in the scene. This section explains how the design and the provided open-source implementation help to answer the queries by
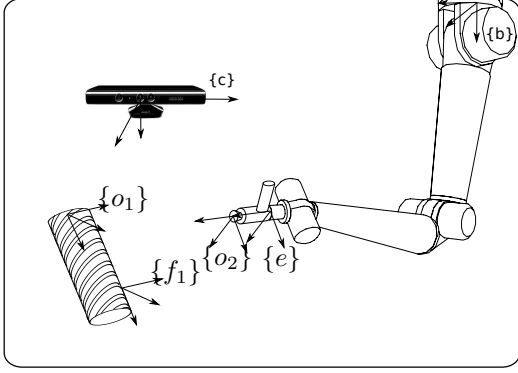
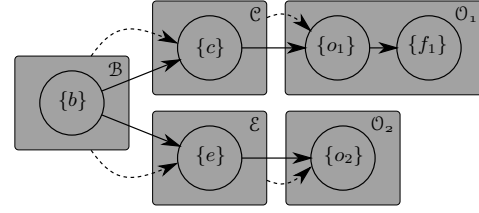Fig. 7: Robot performing a spray painting operation

Fig. 8: Scene graph for example of robot performing spray painting operation. Pose and twist relationships are indicated with solid and dashed arrows, respectively.

means of an application example. The proposed example highlights a typical work flow when using scene graphs in robotics.

The example involves a robot that spray paints a cylindrical object, as illustrated in Figure 7. The cylindrical object is moving in the environment, while being observed by a point cloud camera. The robot holds the spray gun. To complete the painting task, the robot program has to determine the joint angles and joint angle velocities of the robot holding the spray gun such that the desired spray-painting behavior is obtained. In order to control the spray-painting behavior the relative pose and twist between the spray gun and cylindrical object have to be obtained.

When solving geometry problems one first has to identify the rigid bodies and the frames attached to them. In the example we have: 1) $\{b\}$ attached to the base $\mathcal{B}$ of the robot, 2) $\{e\}$ attached to the end-effector $\mathcal{E}$ of the robot, 3) $\{o_2\}$ attached to the spray gun $\mathcal{O}_2$, 4) $\{o_1\}$ attached to the cylindrical object $\mathcal{O}_1$, 5) $\{f_1\}$ coinciding with the desired spray-painting position on the cylindrical object $\mathcal{O}_1$, and 6) $\{c\}$ attached to the point cloud camera $\mathcal{C}$.

In the example the following poses are available: 1) PoseCoord$(\{c\}|\mathcal{C}, \{b\}|\mathcal{B}, [b])$ determined by the mounting of the point cloud data with respect to the robot base, 2) PoseCoord$(\{o_1\}|\mathcal{O}_1, \{c\}|\mathcal{C}, [c])$ determined by an object detection algorithm using the point cloud data, 3) PoseCoord$(\{f_1\}|\mathcal{O}_1, \{o_1\}|\mathcal{O}_1, [o_1])$ determined by the desired spray-painting position on the cylindrical object, 4) PoseCoord$(\{o_2\}|\mathcal{O}_2, \{e\}|\mathcal{E}, [e])$ determined by the mounting of the spray gun on the robot end-effector, and 5) PoseCoord$(\{e\}|\mathcal{E}, \{b\}|\mathcal{B}, [b])$ determined by the current robot joint position. Furthermore, the following twists are available: 1) TwistCoord$(c|\mathcal{C}, \mathcal{B}, [b]) = \mathbf{0}$ determined by the fixed mounting of the point cloud data with respect to the robot base, 2) TwistCoord$(o_1|\mathcal{O}_1, \mathcal{C}, [c])$ determined by an object detection algorithm using the point cloud data, 3) TwistCoord$(o_2|\mathcal{O}_2, \mathcal{E}, [o_2]) = \mathbf{0}$ due to the rigidly assumed mounting of the spray gun on the robot end-

effector, and 4) TwistCoord$(e|\mathcal{E}, \mathcal{B}, [e])$ determined by the current robot joint velocities.

Listing 1 shows how the scene graph is created and how poses and twists are added. Remark that the velocity of the object as measured by the point cloud camera TwistCoord$(o_1|\mathcal{O}_1, \mathcal{C}, [c])$ is not a screw twist (since its orientation frame $[c]$ does not belong to the frame $\{o_1\}$ attached to the body $\mathcal{O}_1$ (with origin the velocity reference point o_1)). Therefore, when adding the twist to the scene graph, the pose relationships will be used to automatically transform the twist into a screw twist. Figure 8 shows the resulting scene graph for the example. Finally, Listing 2 shows how the scene graph can be queried for poses and twists.

Listing 1: Creating the scene graph - C++

```cpp
// Create scene graph
SceneGraph<Vector,Rotation,Vector,Vector>* ←
    scene_graph = new SceneGraph<Vector,←
    Rotation,Vector,Vector>();
// Adding a pose
Rotation coordinatesRot=EulerZYX(M_PI/4,0,0);
Orientation<Rotation> orient(←
    OrientationCoordinatesSemantics("f1","O1","←
    o1","O1","o1"),coordinatesRot);
Vector coordinatesPos(0.02,0,0.2);
Position<Vector> pos("f1","O1","o1","O1","o1",←
    coordinatesPos);
Pose<Vector,Rotation> pose(pos,orient);
scene_graph->AddPose(pose);
// Adding a twist
Vector coordLinVel(0.2,0.1,0.1);
LinearVelocity<Vector> linVel("o1","O1","C","c"←
    ,coordLinVel);
Vector coordRotVel(0.05,0.1,0.2);
AngularVelocity<Vector> rotVel("O1","C","c",←
    coordRotVel);
Twist<Vector,Vector> twist(linVel,rotVel);
scene_graph->AddTwist(twist);
```

Listing 2: Queries on the scene graph - C++

```cpp
//Querying for pose
PoseCoordinatesSemantics poseQuery = ←
    PoseCoordinatesSemantics("f1","f1","O1","o2←
    ","o2","O2","o2");
Pose<Vector,Rotation> poseFound;
```

```
scene_graph->GetPose(poseQuery,poseFound);
//Querying for twist
TwistCoordinatesSemantics twistQuery = ←
    TwistCoordinatesSemantics("f1","O1","O2","←
    o2");
Twist<Vector,Vector> twistFound;
scene_graph->GetTwist(twistQuery,twistFound)
```

VI. Discussion and Conclusions

This paper presents a scene graph explicitly supporting and relying on the geometric semantics. As the geometric semantics software implements checks of all queries, its use will eliminate a source of often hard to track down errors. We believe that by relying on the geometric semantics library for building scene graphs, the (robotics) researcher is given a tool that makes working with geometric relationships easier and much less prone to errors.

Compared to tf2, 1) The presented scene graph allows for any query on the relative position, orientation, pose, linear velocity, angular velocity, or twist between rigid objects in the scene. 2) By relying on the geometric semantics the scene graph naturally allows to treat coordinate frames that are fixed with respect to each other, i.e. attached to the same body. 3) The templated design and use of geometric semantics allow the user the freedom to choose whatever coordinate representation he thinks is most suited, without the need for coordinate conversions leading to degraded performance. 4) Furthermore, the geometric semantics check if constraints imposed by the particular coordinate representation are met during construction and query resolving. If not, meaningful errors are given. 5) Finally, the proposed graph relies on the boost graph library, hereby reusing existing support for graph operations, in stead of reimplementing it. However, as opposed to tf2, there is no support for distributed deployment, nor is there (yet) support for taking into account time.

The constraints of the proposed minimal scene graph can cause pose and twist relationships to be rejected (see Section IV). Therefore, the information contained in the rejected relationships is lost. Future work will explore automatic strategies that can "transform" the rejected pose and twist information such that it is accepted by the minimal scene graph. As an example consider Figure 4 where PoseCoord $(\{c2\}|\mathcal{C}, \{f\}|\mathcal{F}, [f])$ is rejected, by inverting this pose however the resulting pose PoseCoord $(\{f\}|\mathcal{F}, \{c2\}|\mathcal{C}, [c2])$ can be added to the minimal scene graph. By doing this automatic transformation, the information in PoseCoord $(\{c2\}|\mathcal{C}, \{f\}|\mathcal{F}, [f])$ can still be added to the scene graph.

Furthermore, future work will introduce time support, uncertainty in the scene graphs, and kinematic chains. The time support will use 1) the twist relationships to obtain pose relationships for the next time step by integration and 2) the pose relationships of subsequent time steps to get twist by differentiation. Uncertainties can help to take into account e.g. sensor limitations and measurement noise. Support for kinematic chains allows to include complete robot systems in the graphs.

References

[1] T. De Laet, S. Bellens, R. Smits, E. Aertbeliën, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies: Semantics for standardization," *IEEE Robotics and Automation Magazine*, vol. 20, no. 1, pp. 84–93, 2013.

[2] T. De Laet, S. Bellens, and H. Bruyninckx, "Semantics underlying geometric relations between rigid bodies in robotics," https://retf. info/rrfcs/0005, 2012, last visited September 2012.

[3] T. De Laet, S. Bellens, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies: from semantics to software," *IEEE Robotics and Automation Magazine*, vol. 20, no. 2, pp. 91–102, 2013.

[4] T. De Laet and S. Bellens, "Geometric semantics software," http://www.orocos.org/wiki/geometric-relations-semantics-wiki, 2012, last visited September 2012.

[5] A. Bar-Zeev, "Scenegraphs: Past, present, and future," http://www. realityprime.com/articles/scenegraphs-past-present-and-future, september 2012.

[6] "Robotics library," http://sourceforge.net/apps/mediawiki/roblib.

[7] http://www.openscenegraph.org/projects/osg/wiki/Support/ Contributors, "OpenSceneGraph," http://www.openscenegraph.org, last visited March 2013.

[8] T. Foote, E. Marder-Eppstein, and W. Meeussen, "tf," http://ros.org/ wiki/tf, 2011, last visited 2012.

[9] T. Foote, W. Meeussen, and E. Marder-Eppstein, "tf2," http://ros. org/wiki/tf2, 2011, last visited 2012.

[10] T. Foote, "tf: the transform library," in *Proceedings of the IEEE International Conference on Technologies for Practical Robot Applications*, Woburn, Massachusetts, USA, 2013.

[11] T. Rühr, "uncertain_tf," http://ros.org/wiki/uncertain_tf, 2013, last visited 2013.

[12] Boost, "The boost graph library," http://www.boost.org/doc/libs/1_ 51_0/libs/graph.

[13] lemon, "Library for Efficient Modeling and Optimization in Networks," https://lemon.cs.elte.hu/trac/lemon, 2012, last visited 2012.

[14] LEDA, "LEDA," http://www.algorithmic-solutions.com/leda, 2012, last visited 2012.

[15] T. De Laet, "Geometric scene graph software," http://git.mech. kuleuven.be/?p=robotics/pose_graph.git, 2012, last visited March 2013.

[16] R. Smits, H. Bruyninckx, and E. Aertbeliën, "KDL: Kinematics and Dynamics Library," http://www.orocos.org/kdl, 2001, last visited August 2012.