Hochschule Zittau/Görlitz

Praktikumsbeleg

Untersuchung und Implementierung von Methoden der CAD-gestützten Roboterprogrammierung

Dongliang Cao

Bearbeitungszeitraum	3 Monate
Matrikelnummer	217043
Betreuer der Ausbildungsfirma	Dr.-Ing. habil. Fan Dai
Gutachter der Hochschule	Prof. Dr. Stefan Bischoff

4. Juni 2019

Danksagung

Ich möchte mich bei Prof. Stefan Bischoff bedanken, dass er mein Betreuer an der Hochschule ist, und mich bei dem Praktikum unterstützt.

Ich möchte auch Dr. Dai Fan, meinem Betreuer im ABB Forschungszentrum danken. In meinem Praktikum hat er mir bezüglich theoretischer Fachkenntnisse und auch im praktischen Anwendungsbereich viel Unterstützung gegeben. Er hat mir alles eindeutig erklärt mit viel Geduld und Verständnis.

Selbständigkeitserklärung

Ich versichere hiermit, dass ich meine Praxisarbeit mit dem Thema „Untersuchung und Implementierung von Methoden der CAD-gestützten Roboterprogrammierung“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Mannheim, 4. Juni 2019
Ort, Datum

Dongliang Cao
Unterschrift

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die zunehmende Komplexität von Produkten und Maschinen sowie kurze Produktionszyklen bei kleinen Losgrößen stellen die Industriebranche vor große Herausforderungen. Sowohl die Programmierung von Industrierobotern im Online-Modus mit Handbediengeräten als auch im Offline-Modus mit virtueller Simulation erfordert spezielle Kenntnisse in der Robotik und in fertigungsabhängigen Robotersteuerungssystemen. Bei komplizierten Aufgaben ist diese sehr zeitaufwändig. Insbesondere werden klein- und mittelständische Unternehmen mit zusätzlichen Hindernissen konfrontiert wie hohe Investitionen für die Ausbildung der nicht qualifizierten Mitarbeiter und die, wegen der Inbetriebnahme von Roboterzeller, verlängerten Produktionszyklen.

1.2 Motivation

Um die Ingenieure von den wiederholenden, eintönigen und zeitaufwändigen Programmierverfahren zu befreien, damit sie sich mehr auf das Projekt selbst und dessen notwendigen Funktionen konzentrieren können, benötigt es neue Methoden für die Erzeugung des Roboterprogrammes, die effizienter, intuitiver und benutzerfreundlicher ist.

Grundsätzlich Offline-Programmierung (OLP), im Vergleich zur Online Programmierung kann dank der Simulationsumgebung mehr Zeit und Bemühung für den Benutzer sparen. Der Benutzter braucht nicht vor Ort den echten Roboter zu steuern, um die gewünschte Position zu erreichen, sondern am Bildschirm im Büro die notwendigen Positionen, Konfigurationen und Befehle in die Simulationsumgebung eingeben. Der Mangel an heutiger Offline-Software erklärt sich auch dadurch, dass, wenn der Benutzer eine lange Reihe von Bewegungsanweisungen definieren muss, er entweder viele Zeit benötigt, um jede Position zu definieren oder die genauen Positionen eventuell nicht bekannt sind.

Die meisten Informationen über die wichtigsten Positionen für die Bewegungsanweisungen beinhaltet normalerweise das Objekt, mit dem der Roboter interagiert. Die Informationen über das Objekt können durch die Analyse der notwendigen geometrischen Daten genutzt werden. Leider gibt es bisher wenige Ansätze für die Kombination von CAD-Daten und Offline Programmierung.

Aus obigen Gründen möchte ich in meinem Praktikum ein Add-In-Programm in Robotstudio entwickeln, um die automatisch Erzeugung der nötigen Informationen für den Einschubprozess von zwei Objekten zu ermöglichen.

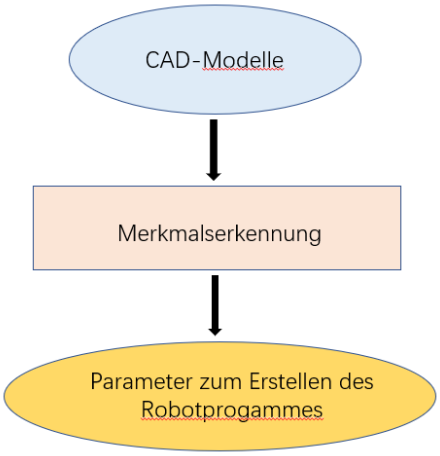


Abbildung 1: Bearbeitungsverfahren für CAD-Daten

1.3 Umsetzungsprozess

Die Realisierung des Prozesses beinhaltet:

Extraktion und Analyse der Informationen einer CAD-Datei Erwerb der geometrischen Informationen und Topologieinformationen des Objekts aus der CAD-Datei

Filtern der Informationen einer CAD-Datei Ausschluss der irrelevanten Informationen und Herausfiltern der Informationen, die Beziehungen zwischen Objekten enthalten

Festlegung der Einschubrichutung und des Einschubabstandes Kalkulation und Bestimmung der potentiellen Einschubrichtungen und entsprechenden -distanzen mithilfe von gefilterten geometrischen Informationen und Topologieinformationen

Visualisierung der Ergebnisse Visualisierung der Ergebnisse in der Benutzeroberfläche von RobotStudio

2 Theoretische Grundlangen

2.1 Offline Programmierung

2.1.1 Definition

Die Offline-Programmierung (OLP) ist eine Roboterprogrammierungsmethode, bei der das Roboterprogramm unabhängig von der eigentlichen Roboterzelle erstellt wird. Bei der Offline-Programmierung wird die Roboterzelle durch ein grafisches 3D-Modell in einem Simulator dargestellt. Heutzutage helfen OLP Roboterintegratoren dabei, die optimalen Programme für den Roboter zu erstellen, um eine bestimmte Aufgabe auszuführen. Bei der Simulation des Roboterprogramms können Roboterbewegungen, Erreichbarkeitsanalysen, Kollisions- und Beinahe-Erkennung sowie Zykluszeitberichte berücksichtigt werden[1].

Vorteile

1) Die Offline-Programmierung unterbricht die Produktion nicht, da das Programm für den Roboter außerhalb des Produktionsprozesses auf einem externen PC geschrieben wird.

2) Integratoren und Endbenutzer können beim Entwurf einer Arbeitszelle Zeit und Geld sparen im Vergleich zur Online-Programmierung.

3) Die Fähigkeit zu analysieren, wie sich eine Arbeitszelle verhält, bevor Zeit und Geld in Geräte investiert werden, sorgt für eine reibungslose Umsetzung vom Konzept zur Realität.

Vorgehensweise der Offline-Programmierung

1) **Erstellung der Arbeitszelle**
   Eine Arbeitszelle bzw. Roboterzelle bezieht sich auf eine Kombination von einem oder mehreren Robotern und den damit verbundenen Werkzeugen, andere Werkstücke und Vorrichtungen. Wenn man eine Arbeitszelle erstellt, sollte man zuerst alle Komponenten in die Arbeitszelle importieren und danach platzieren. Außerdem muss der Roboter mit einer virtuellen Steuerung verbunden werden, um programmieren zu können.Ein Werkzeug ist ein spezielles Objekt (z.B. eine Lichtbogenschweißzange oder ein Greifer), das an einem Werkstück arbeitet. Für korrekte Bewegungen in Roboterprogrammen müssen die Parameter des Werkzeuges in den Werkzeugdaten angegeben werden. Der wesentliche Teil der Werkzeugdaten ist der Werkzeugarbeitspunkt (TCP).

2) **Programmierung von Robotern**
Bevor der Benutzer ein Programm für seinen Roboter erstellt, sollte er die vorher genannte Arbeitszelle einrichten, in der sein Roboter arbeiten soll, einschließlich des Roboters, der Werkzeuge und Vorrichtungen.

Das Verfahren für die Programmierung von Robotern kann in 5 Schritte unterteilt werden.

- Erstellen von Positionen und Bahnen
- Prüfung der Positionsorientierung und Erreichbarkeit
- Synchronisieren des Programms mit der virtuellen Steuerung
- Ausführen von textbasierter Bearbeitung
- Kollisionserkennung

3) **Simulieren von Programmen**
Mit Simulationen werden vollständige Roboterprogramme auf einer virtuellen Steuerung ausgeführt. Durch Simulation kann die Zykluszeit berechnet, Kollisionen erkannt werden, die E/A-Signale simuliert und auch die Ereignisse (Aktion mit einem Trigger verbunden) behandelt werden, um festzustellen, ob das Robotersystem die Erwartung des Endbenutzers erfüllt.

4) **Ladung des Programmes in die reale Steuerung**
Wenn die Simulation erfolgreich durchgeführt wurde, kann der Benutzer das automatisch generiert Programm von dem Computer in die Steuerung des realen Roboters laden.

2.1.2 Robotstudio

Robotstudio ist ein typisches Anwendungsbeispiel für die OLP, das von ABB entwickelt und unterstützt wird. RobotStudio ermöglicht dem Benutzer das Arbeiten mit einer Offline-Steuerung. Dabei handelt es sich um eine virtuelle IRC5-Steuerung, die lokal auf dem PC ausgeführt wird. RobotStudio basiert auf dem so genannten Virtual Controller, einer exakten Kopie der Originalsoftware, die den Roboter in Produktionsprozessen steuert. So sind realistische Simulationen möglich, denn zum Einsatz kommen die Daten und Konfigurationen, die auch in der realen Produktion zum Einsatz kommen[2]. In meiner Arbeit, wird Robotstudio als eine Benutzeroberfläche zur Interaktion und Visualisierung von generierten Ergebnissen genutzt. Außerdem wird das Robotstudio Software Development Kit (SDK) als Programmbibliothek für die Entwicklung des Add-In-Programmes verwendet.

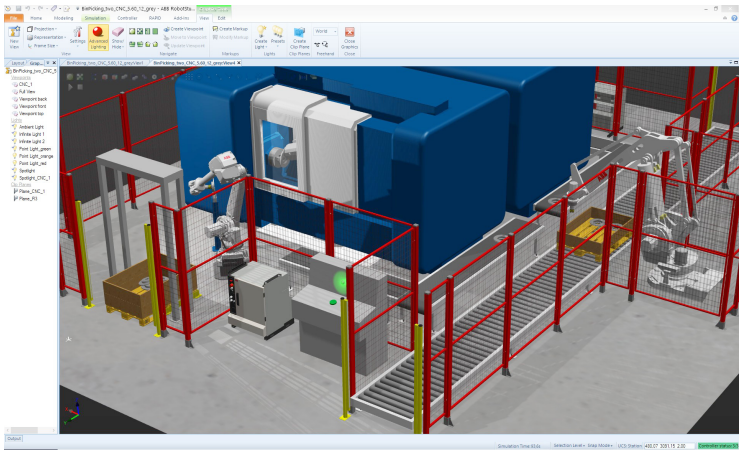


Abbildung 2: Roboterzelle in Robotstudio

2.2 Geometrie und Topologie

Geometrie und Topologie sind die zwei wichtigsten Begriffe für die mathematische Darstellung der räumlichen Informationen von Objekten in der realen Welt. Die Geometrie bezieht sich in dieser Arbeit besonders auf die euklidische Geometrie, die sich mit Punkten, Geraden, Ebenen, Abständen und Winkeln beschäftigt, sowie den Begriffsbildungen und Methoden, die im Zuge einer systematischen und mathematischen Behandlung dieses Themas entwickelt wurden[3]. Neben der mathematischen Beschreibung der räumlichen Lage und Form von einzelnen Komponenten, beschreibt die Topologie die Lagebeziehung zwischen Geoobjekten, wie z.B. Knoten, Kanten und Maschen. In einfachen Systemen entsprechen Punkten, Knoten, Linien Kanten und Flächen Maschen[4].

In den meisten CAD-Datei wird boundary representation (b-rep), zu Deutsch „Begrenzungsflächenmodell“, als Modellierungsform eines Flächen- oder Volumenmodells verwendet. Ein typisches Beispiel ist eine Fläche wird als ein begrenzter Teil einer Oberfläche angesehen. Eine Kante ist ein begrenzter Teil einer Kurve und ein Scheitelpunkt liegt an einem Punkt. In der Welt des Datenaustauschs definiert the Standard for the Exchange of Product Model data (STEP) auch einige Datenmodelle für b-rep. Die allgemeinen generischen topologischen und geometrischen Modelle sind in der geometrischen und topologischen Darstellung nach ISO 10303-42 definiert[5].

2.2.1 Geometrie

Die Geometrie, die für die Modellierung von Geoobjekten in CAD-Systemen verwendet wird, ist die euklidische Geometrie. Die Geometrie kann man in vier Kategorien einteilen.

1) **Punkte**, die in einem dreidimensionalen Raum existieren.
Der Punkt ist das grundlegendste geometrische Konzept in einem Raum. Ein Punkt im dreidimensionalen Raum wird normalerweise in einem kartesischen Koordinatensystem als eine Position dargestellt und die Position enthält drei Werte, die sich auf x, y, z-Koordinaten beziehen.

Hinweis:
Der Punkt in einem dreidimensionalen Raum kann nicht nur eine Position, sondern auch einen dreidimensionalen Vektor repräsentieren. Ein Vektor ist als eine räumliche Verschiebung von einer Position zu einer anderen Position zu definieren.

2) **Kurven**, die in einem dreidimensionalen Raum existieren.
In der Mathematik ist eine Kurve ein eindimensionales Objekt. Eindimensional bedeutet dabei informell, dass man sich auf der Kurve nur in einer Richtung (bzw. der Gegenrichtung) bewegen kann. Die Kurve können in der zweidimensionalen Ebene liegen(„ebene Kurve“) oder in einem höherdimensionalen Raum[6].

In CAD-Systemen gibt es zwei wesentliche Arten vom Kurven: analytische Kurven und interpolierte Kurven. Alle analytischen Kurven werden von einem Parameter, der normalerweise t genannt wird, parametrisch dargestellt. Die drei wichtigsten Typen der analytischen Kurven sind Geraden, Ellipsen (einschließlich Kreise) und Helices.

- *Geraden*
  Eine Gerade wird durch einen Punkt $\vec{p_0}$ und eine Richtung $\vec{r}$ dargestellt. Eine Gerade kann eine unendliche gerade Linie, eine teilweise begrenzte gerade Linie (ein Strahl) oder eine begrenzte gerade Linie (ein Liniensegment) darstellen

  Im dreidimensionalen Raum kann die Position eines Punktes auf einer Geraden durch die folgende parametrisierte Gleichung berechnet werden.
  $$\vec{p(t)} = \vec{p_0} + t\vec{r}$$

- *Ellipsen*
  Eine Ellipse wird durch einen Mittelpunkt $\vec{p_0}$, einen Einheitsvektor $\vec{n}$, der senkrecht zur Ebene der Ellipse ist, einen Vektor $\vec{M}$, der die Hauptachse der Ellipse (einschließlich der Größe der Hauptachse) repräsentiert, und das Radiusverhältnis α (das Verhältnis der Nebenachsenlänge zur Hauptachsenlänge) vollständig definiert.

Im dreidimensionalen Raum kann die Position eines Punktes auf einer Ellipse durch die folgende parametrisierte Gleichungen berechnet werden.

$$\vec{p(t)} = \vec{p_0} + \vec{M}\cos t + \vec{m}\sin t$$
$$\vec{m} = \alpha(\vec{n} \times \vec{M})$$

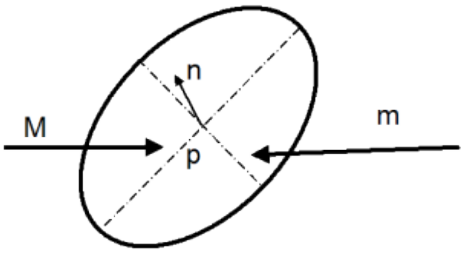


Abbildung 3: Ellipse

– *Helices*
Eine Helix ist typischerweise eine dreidimensionale Spule, die wie ein Schraubengewinde auf der Oberfläche eines Zylinders liegt. Eine Helix wird definiert durch einen Wurzelpunkt, einen Einheitsvektor, der die Achse der Helix definiert, einen Vektor vom Wurzelpunkt zu einem Punkt auf der Kurve, die Steigung, die Drehrichtung und den Parameterbereich.

Im dreidimensionalen Raum kann die Position eines Punktes auf einer Geraden durch die folgende parametrisierte Gleichung berechnet werden.

$$\vec{p(t)} = a\cos t\vec{x_0} + a\sin t\vec{y_0} + bt\vec{z_0}$$

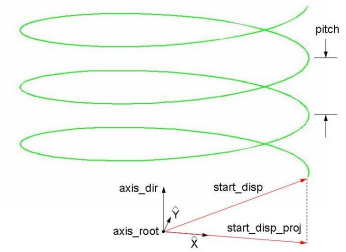


Abbildung 4: Helix

3) **Oberflächen**, die in einem dreidimensionalen Raum existieren.
Der Begriff Oberfläche hat mehrere Bedeutungen. In CAD-Systemen wird dieser Begriff am häufigsten in seinem geometrischen oder mathematischen Sinne verwendet, um ein zweidimensionales Objekt in einem dreidimensionalen Raum mit einer einzigen geometrischen Definition zu beschreiben.

Änlich wie bei Kurven, in CAD-System können Oberfläche im Wesentlichen in zwei Gruppen eingeteilt werden: analytische Oberflächen und interpolierte Oberflächen. Alle analytischen Oberflächen werden von zwei Parametern, die normalerweise u, v genannt werden, parametrisch dargestellt. Die vier wichtigsten Typen der analytischen Oberflächen sind Ebenen, Kegel, Kugel und Torus.

- *Ebenen*
  In der Geometrie repräsentiert eine Ebene eine unendliche ebene Fläche oder einen begrenzten Bereich auf einer solchen Fläche.
  Die Parameter einer Ebene werden durch einen Punkt und einen Normalenvektor definiert. Die Parametrisierung einer Ebene wird durch zwei zusätzliche Parameter definiert: einen Vektor senkrecht zur Normalen, der die U- Parameterrichtung und Skalierung widerspiegelt, sowie eine Flag, die angibt, ob die Parametrisierung der Ebene für Rechts- oder Linkshänder erfolgt.

  Normalerweise wird eine Ebene in Bezug auf das rechtshändige Koordinatensystem definiert $(\vec{U}, \vec{V}, \vec{N})$. Die Richtung von $\vec{U}$ ist von dem entsprechenden VeKtor bestimmt und die Richtung von $\vec{V}$ ist durch $(\vec{N} \times \vec{U})$ bestimmt.

$$p(\vec{u}, v) = \vec{R_0} + u\vec{U} + v\vec{V}$$

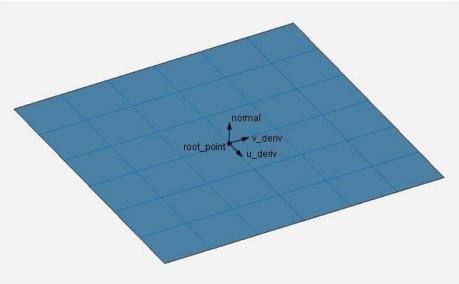


Abbildung 5: Ebene

– *Kegel*
In der Geometrie repräsentiert ein Kegel entweder einen Kegel oder einen Zylinder. Die Geometrie eines Kegels wird durch eine Basisellipse und den Sinus und Cosinus des Haupthalbwinkels des Kegels definiert. Die Normale der Basisellipse repräsentiert die Achse des Kegels.

$$x = au\cos v$$
$$y = au\sin v$$
$$z = u$$

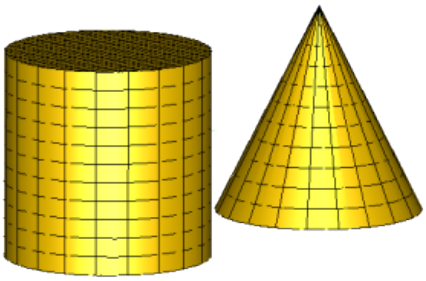


Abbildung 6: Kegel

– *Kugel*
In der Geometrie repräsentiert eine Kugel alle Punkte im dreidimensionalen Raum, die sich im Abstand R von einem festen Punkt dieses Raumes befinden. Der feste Punkt heißt der Mittelpunkt der Kugel und r ist eine reelle Zahl, die als Radius bezeichnet wird.

Die parametrische Gleichung für eine Kugel lautet:

$$p(\vec{u}, v) = \vec{C} + |R|\sin u\vec{P} + |R|\cos u(\cos v\vec{Q} + \sin v\vec{R})$$

$\vec{C}$ ist der Mittelpunkt der Kugel
$\vec{P}$ ist der Polrichtung
$\vec{Q}$ ist die Richtung zum Ursprung des Parameterraums
$\vec{R} = \vec{P} \times \vec{Q}$

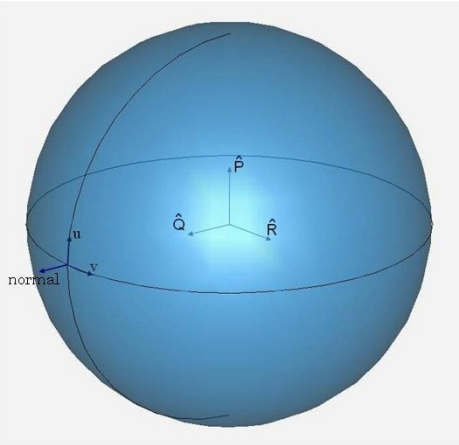


Abbildung 7: Kugel

2.2.2 Topologie

Das Grundkonzept von boundary representation (b-rep) ist die Topologie, die beschreibt, wie Elemente begrenzt und verbunden werden. Topologie beschreibt die Beziehungen zwischen unterschiedlichen Geoobjekten.

Durch Topologie kann, zum Beispiel, ausgedrückt werden, dass eine Kante E_1 durch die Eckpunkte V_1 und V_2 begrenzt ist. Falls wir auch wissen, dass eine andere Kante E_2 durch die Eckpunkte V_2 und V_3 begrenzt ist, dann können wir ableiten, dass die Kanten E_1 und E_2 benachbart sind, weil V_2 beide Kanten begrenzt.

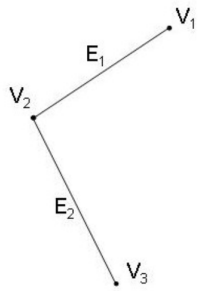


Abbildung 8: Veranschaulichung des Topologie Beispiels

Eine typische Hierarchie für topologischen Elemente mit Volumen besteht aus Körper, nicht verbundene Körper, Hüllen, Subhüllen, Flächen, Umrandungen, gerichtete Kanten, Kanten, Punkten. Für die Elemente ohne Volumen existieren keine Flächen, sondern Drähte.

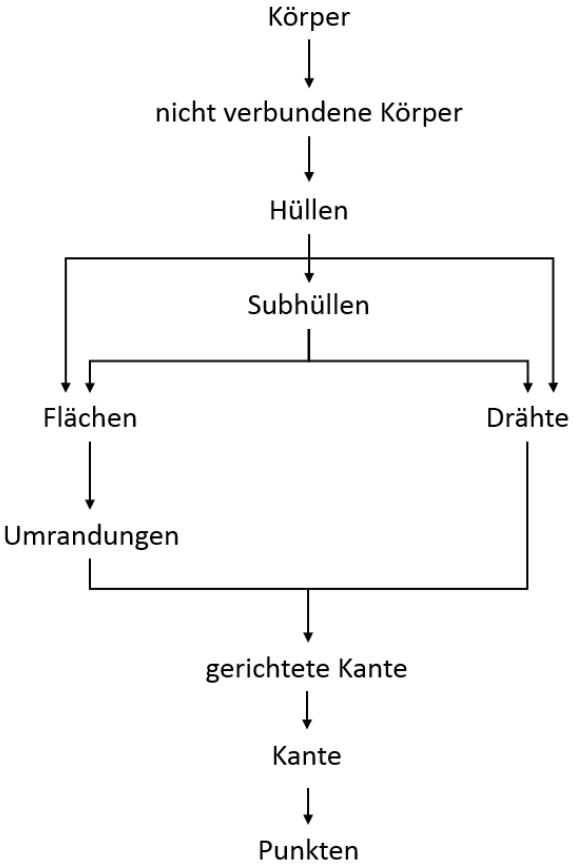


Abbildung 9: hierarchische Beziehungen zwischen topologischen Elementen

2.3 CAD Modellierung

Verschiedene CAD-Softwares unterstützen unterschiedliche CAD-Dateien. In SolidWorks werden beispielsweise ein Teil als ".prt"definiert, und eine Baugruppe als ".asm"gespeichert. Die verschiedenen proprietären CAD-Dateiformate werden von unterschiedlichen CAD-Konstruktionssoftwares wie Pro Engineer, SolidWorks und AutoCAD verwendet. Aufgrund der Vielfalt von CAD-Dateiformaten ist es sehr wichtig, ein neutrales CAD-Dateiformat, mit dem die Daten zwischen verschiedenen CAD-Programmen ausgetauscht werden können, zu finden.

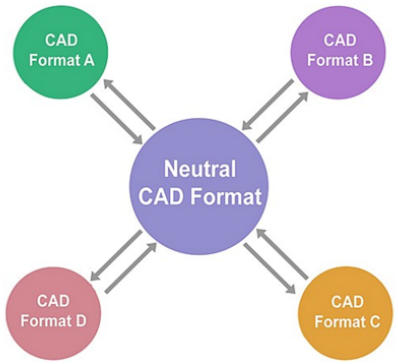


Abbildung 10: neutrales CAD-Dateiformat hilft Datenaustausch

Eines der beliebtesten CAD-Dateiformat ist STEP. Die Vorteile von STEP spiegeln sich hauptsächlich in den folgenden vier Punkten wider.

1) Ermöglicht das Anzeigen und Ändern von Geometrie mithilfe eines beliebigen CAD-Tools, das STEP-Geometrie interpretieren kann, und unterbricht die Abhängigkeit zwischen CAD-Systemen und Produktdefinition

2) Enthält das Assembly-Schema, das alle Connector-Entitäten enthält

3) Nützlich zum Gruppieren von mechanischen Elementen in Bestimmte Ansicht

4) Einfache Ableitung der impliziten Informationsinhalte.

3 Aufgabenstellung

3.1 Addin-Entwicklung in RobotStudio

Die Entwicklung für die Benutzeroberfläche zur Interaktion und Visualisierung wird mit dem RobotStudio SDK ausgeführt. Hierbei handelt es sich um ein Framework, mit dem die ABB RobotStudio Funktionen über die verfügbaren application programm interface (API) bearbeitet werden. Die Programmiersprache C# wird zum Entwickeln der Front-End-Operationen verwendet. Das SDK bietet Visual Studio-Projektvorlagen und APIs zum Erweitern von RobotStudio.

kurze Vorstellung für RobotStudio SDK und arbeitsrelevante Komponenten
RobotStudio SDK wird unter das .NET Framework entwickelt und von C# Programmiersprache geschrieben. Grundsätzlich kann RobotStudio SDK nach unterschiedlichen Namespaces in verschiedenen Komponenten geteilt werden. Am häufigsten benutzte Namespaces sind:

- **ABB.Robotics.Math:** Der Namespace wie den Namen enthält alle Struktur und Klassen für mathematische Operationen in RobotStudio. Einige typische und häufig verwandte Beispiele sind Matrix4 zur Transformation (einschließlich Rotation und Translation), Vector3 zur Beschreibung einer Position, einer Vektor oder Eulersche Winkel und BoundingBox zur Beschreibung der minimale Begrenzungs- oder Umschließungsrahmen für eine Geometrie.

- **ABB.Robotics.RobotStudio:** Der Namespace enthält Klassen, die die grundlegenden Bausteine der RobotStudio Pojektmodellen darstellen, z.B. das Speichern und Laden von Projekten, benutzerdefinierte Attribute, das Rückgängigmachen und verschiedene Maßeinheiten. Viele der Klassen in diesem Namespace werden als Basisklassen für die spezifischen Klassen im ABB.Robotics.RobotStudio.Stations-Namespace genutzt. Ein Beispiel davon ist ProjectObject, das als eine abstrakte Basisklasse für alle Objekte, die Teil eines Projekts sind, definiert wird.

- **ABB.Robotics.RobotStudio.Environment:** Der Namespace enthält Klassen für die Beschäftigung mit der RobotStudio-Benutzeroberfläche, z.B. die Buttons, Windows und Kontextmenüs.

- **ABB.Robotics.RobotStudio.Stations:** Der Namespace enthält Klassen, die das RobotStudio-Objektmodell, das Projektmodell, die Simulationslogik und den Informationsstrom darstellen. Die relevantesten Objektmodelle für meine Arbeit handelt es sich um die geometrische Topologie Klassen. Die Hierarchie für die Topologie kann durch folgende Abbildung eindeutig

dargestellt wird. Die Klassen, die von der Basisklasse „GraphicComponent“ abgeleitet sind, kann direkt in grafische Benutzeroberfläche (GUI) im RobotStudio hinzugefügt und gezeigt werden. Die anderen müssen eine Unterkomponente davon sein.

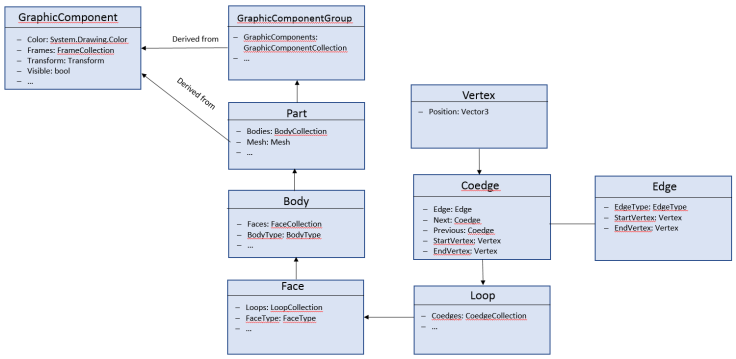


Abbildung 11: geometrische Topologie in RobotStudio

- **ABB.Robotics.RobotStudio.Stations.Forms:** Der Namespace enthält Klassen für die Interaktion mit der 3D-Ansicht sowie eine Reihe von RobotStudio-spezifischen Kontrolle für Formulare und ToolWindows, z.B. Kontrolle für die Eingabe von Koordinaten und die Darstellung von Maßeinheiten.

Die Vorgehensweise für die Entwicklung in RobotStudio

1) **Herunterladen und Installation von RobotStudio SDK**
   RobotStudio SDK kann man in die Website ABB Developer Center herunterladen. Bevor man RobotStudio SDK installiert, muss man beachten, dass zuvor Visual Studio 2015/2017 schon am Computer installiert ist.

2) **Erstellung eines neuen Add-in Projektes in Visual Studio**
   Öffnen Sie zuerst Microsoft Visual Studio und wechseln Sie zu einem neuen Projekt. Aufgrund der Installation von RobotStudio SDK werden in der folgenden Abbildung die folgenden Vorlagen angezeigt. Wählen Sie aus der Liste RobotStudio 6.0 Empty Add-In aus und wählen Sie einen Namen für das Projekt.

   Aus dieser Vorlage wird die folgende Lösung mit der *.cs-Datei und den grundlegenden RobotStudio SDK *.dlls, die als Referenz zu dieser Lösung verwendet werden, generiert,. Außerdem wird mit der *.sln die Umgebung für die Entwicklung des RobotStudio-Add-Ins erstellt.

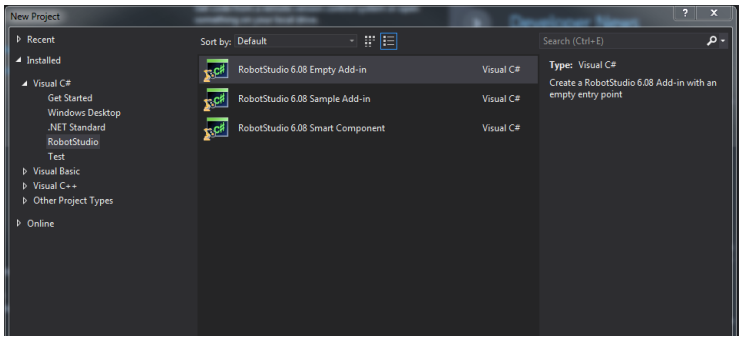


Abbildung 12: neues Projekt für RobotStudio Add-in auswählen

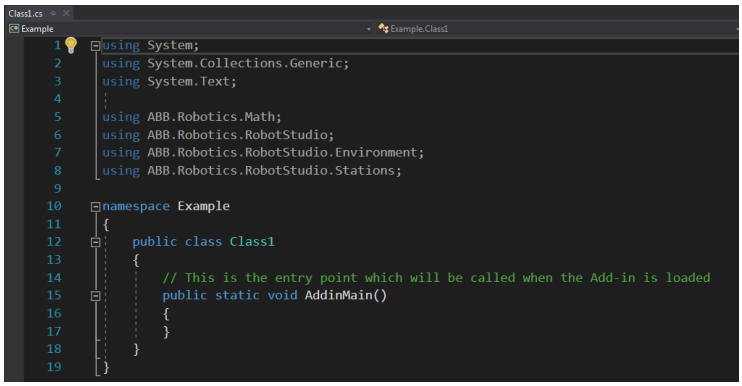


Abbildung 13: leere Lösung für RobotStudio Add-in

Führen Sie den Build-Befehl in Visual Studio aus, um den Code zu kompilieren und eine *.rsaddin-Datei für das Add-In zu generieren. Diese *.rsaddin-Datei ist dafür verantwortlich, dass RobotStudio die Add-In-Assembly (die .dll-Datei) laden kann.

3) **UI-Entwicklung**
Im RobotStudio-Arbeitsbereich gibt es verschiedene Arten von Bereichen, z.B. Ribbonbereich, Eigenschaftsbereich und Arbeitsbereich.

Um sich besser an die Gewohnheit des Benutzers anzupassen, verwende ich den Ribbonbereich und füge ich ein neues RibbonButton hinzu. Die Reihenfolge für die Hinzufügung des Ribbons ist grundsätzlich wie folgendes:

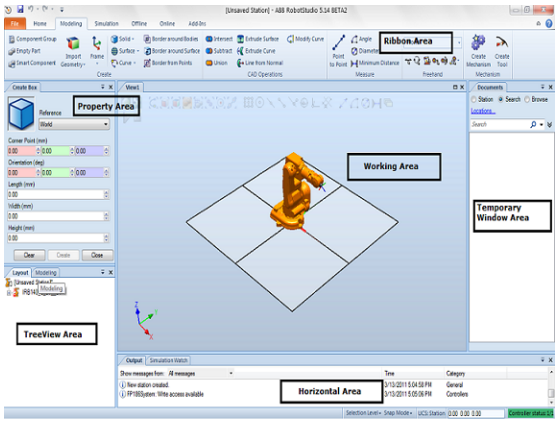


Abbildung 14: Arbeitsbereich in RobotStudio

```
//create a new tab
RibbonTab ribbonTab = new RibbonTab("myTab", "myTab");
UIEnvironment.RibbonTabs.Add(ribbonTab);
//create a new ribbon group
RibbonGroup ribbonGroup = new RibbonGroup("myGroup", "myGroup");
ribbonTab.Groups.Add(ribbonGroup);
//create a new button
CommandBarButton button = new CommandBarButton("myButton", "myButton");
ribbonGroup.Controls.Add(button);
```

Abbildung 15: Der Kode für die Erstellung von einem Button

Erstens, ein neues RibbonTab erstellen und in UIEnvironment hinzufügen.
Zweitens, ein neues RibbonGroup erstellen und in das vorher erstellte RibbonTab hinzufügen.
Drittens, ein neues RibbonButton erstellen und in das vorher erstellte RibbonGroup hinzufügen.

Um die Organisation der Kodestruktur zu verbessern und später besser zu verwalten, habe ich für jeden individuellen Button eine neue Klasse erstellt, sodass die Funktionen und die Daten in dieselbe Klasse eingekapselt sind.

ToolWindow ist eine andere wichtige Klasse für GUI, die multifunktional als ein einziger Button ist. Ein ToolWindow ist ein Container für andere graphische Kontrolle, somit ist ein ToolWindow vielseitig und leicht erweiterbar. Eine typische Anwendung für die Nutzung von ToolWindow ist wie folgende Abbildung gezeigt.

```
class ButtonAddInsert
{
    private ViewInsertDefinition view_InsertDefinition = new ViewInsertDefinition();
    public ButtonAddInsert(RibbonGroup ribbonGroup, RibbonTab ribbonTab)
    {
        CommandBarButton btn_new_action = new CommandBarButton("New_Insert", "Insert");
        btn_new_action.HelpText = "Create a new insert action into the assembly tree";
        btn_new_action.Image = Image.FromFile(Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), "Ri
        btn_new_action.DefaultEnabled = true;

        ribbonGroup.Controls.Add(btn_new_action);

        //Include Seperator between buttons
        CommandBarSeparator seperator = new CommandBarSeparator();
        ribbonGroup.Controls.Add(seperator);

        ribbonGroup.SetControlLayout(btn_new_action, RibbonControlLayout.Large);

        // Add an event handler.
        //btn_new_action.UpdateCommandUI += new UpdateCommandUIEventHandler(Button_UpdateCommandUI_New_Action);
        // Add an event handler for pressing the button.
        btn_new_action.ExecuteCommand += new ExecuteCommandEventHandler(Button_ExecuteCommand_New_Action);
    }
```

Abbildung 16: Ein Beispiel für eine Buttonklasse

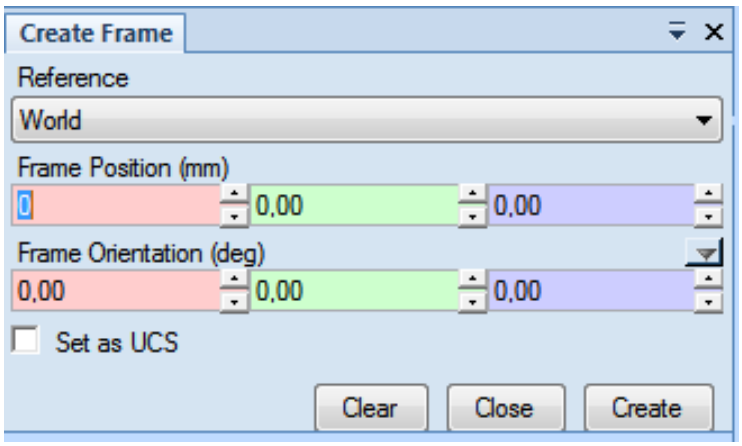


Abbildung 17: Ein Beispiel für die Anwendung von ToolWindow

Das Beispiel wird zur Erzeugung eines neuen Frames genutzt. In diesem ToolWindow kann man unterschiedliche graphische Kontrolle sehen, wie ComboBox, Label, NumericTextBox, Button, die als ein Einheit zusammenarbeiten, um die Position und die Orientierung in Bezug auf unterschiedlichen Referenzframen zu bestimmen und endlich ein neues Frame zu erzeugen.

In meinem kleinen Add-In-Programm, wählt der Benutzer einfach zwei Körper aus eine Geometrie Komponente und danach bestätigt, deshalb sieht das ToolWindow wie folgende Abbildung aus.

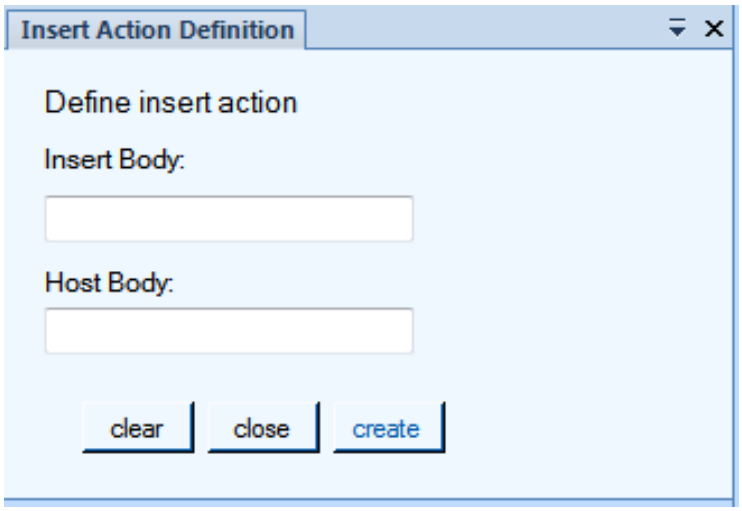


Abbildung 18: Tool Window für die Auswahl von zewi Körpern

Für die Auswahl von den gewünschten Körpern kann der Benutzer entweder in Bereich von Baumansicht oder in Arbeitsbereich durch Mausklick ermöglichen. Um diese Funktion zu verwirklichen, muss man ein neues Ereignis „ObjetAdded“ unter dem Objekt „ProjectSelection“, das als eine Eigenschaft unter der Klasse „Selection“ ist, registrieren. Nach dem Kicken von „Create Button“ werde die Analyse von zwei ausgewählten Körpern beginnen und endlich die kalkulierte Einschubrichtung und -abstand zwischen zwei Körpern in Arbeitsbereich zeigen.

```
//add select body event
Selection.SelectedObjects.ObjectAdded += SelectedObjects_ObjectAdded;
```

Abbildung 19: Registrierung eines neuen Ereignisses

```
private void SelectedObjects_ObjectAdded(object sender, SelectionEventArgs e)
{

    if (!this.toolWindow.Visible || !this.toolWindow.DefaultTabVisibility) return;
    if(e.SelectionObject is Body b)
    {
        if (this.selectInsert)
        {
            this.InsertBodyTB.Tag = b;
            this.InsertBodyTB.Text = b.Parent.DisplayName + "-" + b.DisplayName;
            this.selectInsert = false;
        }
        else
        {
            this.HostBodyTB.Tag = b;
            this.HostBodyTB.Text = b.Parent.DisplayName + "-" + b.DisplayName;
            this.selectInsert = true;
        }
    }

}
```

Abbildung 20: Inhalt des Ereignises

3.2 CAD-Modell Analyse Mithilfe von ACIS Modeler

3.2.1 Kurze Vorstellung von ACIS Modeler

Der 3D ACIS Modeler (ACIS) ist eine objektorientierte dreidimensionale (3D) Geometrie- / Volumenmodellierungs-Engine von Spatial Corp. Sie dient als Grundlage jeder 3D-Modellierungsanwendung für Endbenutzer. ACIS wurde in C ++ geschrieben und bietet ein offenes Architektur-Framework für die Modellierung von Drahtmodellen, Oberflächen und Volumenkörpern aus einer gemeinsamen, einheitlichen Datenstruktur.

3.2.2 Vorstellung der angewandten Funktionen

Für meine Analyse werde ich wesentlich die Modellierungskomponente und die Komponente für die Analyse von Beziehung der Objekte benutzt.

- **Modellierungskomponent:** Die Modellierungskomponente lässt sich wesentlich als zwei Gruppe einteilen.

  Die erste Gruppe beinhaltet alle Klassen und Strukturen, die für die Beschreibung und Modellierung der Geometrie und Topologie verantwortlich sind. Für die Topologie, die hierarchische Struktur von ACIS ist ähnlich wie die in obergenannte Struktur von RobotStudio nur ohne die oberen Klassen wie Part und GraphicComponentGroup, die als eine Visualisierungseinheit genutzt werden. Aber für die Geometrie, bietet ACIS andere separat geometrische Klassen an. Die Trennung von Geometrie und Topologie ermöglicht effizientere Algorithmen und bessere Modifikation von Geometrie. Jedes geometrische Objekt versteckt in ein topologisches Objekt, d.h. kann der Programme durch einen Pointer im topologischen Objekt das geometrische Objekt ergreifen. Zum Beispiel, das FACE ist ein topologisches Objekt in ACIS, das eine begrenzte Oberfläche repräsentiert, und dahinter verborgenes geometrisches Objekt SURFACE wird durch einen Pointer in FACE zugewiesen.

  Die zweite Gruppe beinhaltet viele nützliche API, die hilfreich für die Traverse durch ganze topologische Hierarchie sind. Die APIs vereinfacht die Arbeit von Programmen zu einem großen Teil. Zum Beispiel, wenn der Programme eine Operation für alle Ecken unter dem gleichen Körper durchführen möchte, muss er zuerst alle Ecken unter dem Körper bekommen, was sehr mühsam ist, weil der Programme zuerst alle Oberflächen, danach alle Schleife, endlich alle Kanten durchqueren muss. Glücklicherweise, bieten viele einfach Funktionen für solche Operationen auf unterschiedlichen topologischen Niveaus.

- **Komponente für die Analyse von Beziehung der Objekte** ACIS bietet verschiedene Mechanismen zum Analysieren der Objektbeziehung zwischen zwei Objekten. Es bietet allgemeine Funktionen für den Abstand von Objekte auf unterschiedlichen Niveaus, wie Körper und Körper, Körper und Fläche. Außerdem bietet es alternative Funktionen, die eine facettierte Annäherung an die Objekte verwenden, um den Mindestabstand zwischen ihnen zu bestimmen. Wenn weitere Informationen zu potenziellen Kollisionsbereichen benötigt werden, stellt ACIS eine Kollisionsfunktion bereit, um zu bestimmen, wie zwei Objekte interagieren. Darüber hinaus bietet ACIS eine Reihe von Funktionen, die die Einschlussbeziehung zwischen zwei Objekten oder zwischen einem Punkt und einem Objekt bestimmen.

3.2.3 Die Verbindung zwischen RobotStudio und ACIS

Wie oben vorgestellt wird, RobotStudio SDK ist in C# geschrieben und gleichzeitig ACIS ist in C++ geschrieben. Deshalb müssen wir eine Lösung finden, um in einer gemeinsamen .NET-Lösung die Funktionen zwischen diesen beiden Sprachen zu kommunizieren.

Einer der beste Lösungen für die Kommunikation zwischen beiden Programmiersprachen ist, dank der Kompatibilität von .NET, die Anwendung von C++/CLI. Die common language infrastructure (CLI) wird als eine von Microsoft entwickelte Variante der Programmiersprache C++ betrachtet, die den Zugriff auf die virtuelle Laufzeitumgebung der .NET-Plattform mit Hilfe von speziell darauf zugeschnittenen Spracherweiterungen ermöglicht. C++/CLI erfüllt die von Microsoft entwickelte Spezifikationen zur Sprach- und Plattform-neutralen Entwicklung und Ausführung von .NET-Anwendungen. Programme, die in C++/CLI geschrieben sind, können vom Compiler in common intermediate language (CIL) übersetzt und auf der virtuellen Maschine der .NET-Plattform betrieben werden[7].

Um sich die Referenzklassen in C#, die automatisch von garbage collection (GC) verwaltet werden, und die Klassen in C++ voneinander zu unterscheiden, benutzt CLI eine andere Syntax für die Deklaration und Definition von den Referenzklassen in C#. Ein typischer Unterschied ist wie folgende Abbildung gezeigt.

```
array<int>^ numbers = gcnew array<int>(100);

String^ name;
```

Abbildung 21: Deklaration von Array und String in CLI

3.3 Einschubprozesserkennung

Wenn man über die Montage zwischen zwei Objekten nachdenkt, fällt ihm als Erstes die einfachste und gebräuchlichste Art von Montage ein, d.h. Einschub. Deshalb fokussiere ich in meine Praktikumsarbeit für den Einschub in einen Montageprozess.

3.3.1 Definition des Einschubprozesses

Die erste Frage für die Analyse des Einschubprozesses ist wie kann man den Einschubprozess vollständig definieren, sodass der Prozess bei Generierung des Roboterprogrammes direkt benutzt werden kann. Zur Verallgemeinerung und Vereinfachung des Prozesses kann man annehmen, dass der Einschubprozess ist nur von der Start-, Endposition und der Richtung abhängig. Außerdem der Einschub ist auf eine lineare Translationsbewegung beschränkt und deshalb kann die Start- und Endposition durch einen Abstand ersetzt werden.

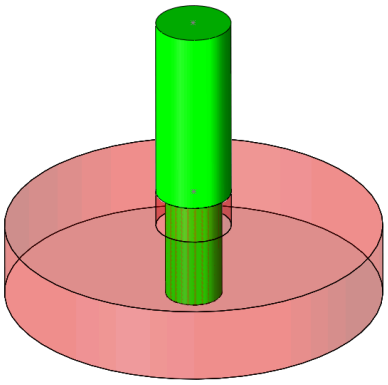


Abbildung 22: Ein Beispiel von Einschubprozess

3.3.2 Definition des Modells in CAD

Die zweite Frage ist wie kann man das CAD-Modell, das relevante Objekte für den Einschubprozess beinhaltet, definieren. Wenn man den Einschubprozess einordnen möchte, ist die Anzahl der beteiligten Teile ein wichtiges Kriterium. Nach diesem Kriterium kann man den Prozess in zwei Arten teilen, nämlich eins-zu-eins und eins-zu-mehr.

Wenn wir eins-zu-mehr Baugruppe zerlegen, können wir feststellen, dass ein bewegliches Teil eine individuelle Montagebeziehung zwischen anderen Teilen in der Baugruppe hatte. Die Lösung für die mehrteilige Montage kann erreicht werden, indem ein bewegliches Teil entnommen und die Beziehung für jeden

festen Teil nacheinander ermittelt wird. Dann können wir das einzelne Ergebnis zwischen einem beweglichen und einen festen Teil akkumulieren und den Montageprozess nacheinander durchführen. Gleichzeitig muss man beachten, dass die Reihenfolge eine wichtige Rolle spielt.

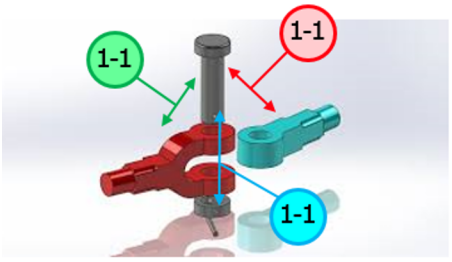


Abbildung 23: Ein Beispiel für eins-zu-mehr Baugruppe

Aus obengenannten Gründen, können wir das Modell so definieren:

- Die Anzahl der Montageteile beträgt zwei
- Ein Montageteil gilt als fest und ein anderes als beweglich
- Anfangszustand von Montageteile ist montiert

3.3.3 Die Lösung für Einschubrichtungserkennung

Die Grundidee

- Die Auswahl von relevantesten Flächen in Festkörper und Einschubkörper separat.
- Extrahierung der möglichen Einschubrichtungen aus den ausgewählten Flächen.
- Festlegung der potentialsten Einschubrichtungen, indem der bewegliche Körper entlang der Richtung bewegen und eine Kollision zwischen beiden Körpern berechnen.
- Berechnung der Einschubdistanz aus die festgelegten Einschubrichtungen.

Die Auswahl von relevanten Flächen spielt eine entscheidende Rolle für die Analyse von Einschubprozess zwischen zwei Objekten. Wenn zu wenig Flächen gefunden werden, kann die Fläche, die die Einschubrichtung enthält, vermissen. Wenn zu viel Flächen gefunden werden, ist die Effizienz der folgenden Analyse sehr niedrig. Deshlb kommt es zu dieser Frage, wie kann man die relevantesten Oberflächen definieren und herausfinden, sodass keine wichtigen Informationen für die Einschubrichtung übersehen werden, gleichzeitig keine überflüssigen Informationen enthalten sind.

- **Entity Clash Calculations**
  Basierenden auf dieser Idee, habe ich zuerst die APIs für „Entity Clash Calculations“ benutzt. Die Eingänge für diese Funktion sind zwei Körper und die Ausgänge sind die zusammengestoßene Flächenpaare und deren Kollisionstypen. Die Vorteile für diese Funktion sind eindeutig. Auf diese Weise erhalte ich nicht nur die Kollisionsflächenpaare, sondern auch den Kollisionstyp, die sehr nützlich ist, um nicht verwandte Flächenpaare auszuschließen. Weil ich mich nur für die Flächenpaare interessiere, deren Normalvektoren einander entgegengesetzt sind.

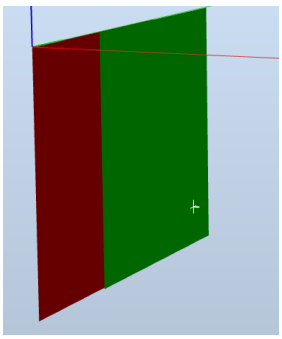


Abbildung 24: Eine Flächenpaare mit entgegengesetzten Normalvektoren

Trotz der Vorteile des Algorithmus habe ich bei Anwendung in verschiedenen CAD-Dateien festgestellt, dass, aufgrund der Toleranz- oder Abstandunterschied in Festkörper und Einschubkörper in unterschiedlichen CAD-Dateien, viele verwandte Flächen, die die Informationen über die Einschubrichtung enthalten, nicht herausgefunden werden können. Einer der wichtigsten Gründ in der Praxis für den Fehlschlag des „Entity Clash Calculations“ Algorithmus ist aufgrund der Präzision verschiedener CAD-Modelle der Mindestabstand bzw. der Grenzwert für die Bestimmung, ob zwei Körper zusammengestoßen sind, sehr unterschiedlich. Beispielsweise kann bei einer großen mechanischen Baugruppe der Mindestabstand zwischen Objekten einige Millimeter betragen, bei einem Präzisionsinstrument kann der Abstand zwischen Objekten jedoch nur einige Mikrometer betragen.

- **Entity Distance Calculation**
  Um obengenanntes Problem zu bewältigen, muss man den Mindestabstand in unterschiedlichen CAD-Modellen berücksichtigen. Deshalb habe ich stattdessen die APIs für „Entity Distance Calculation“ benutzt. Die verwandten Flächen nach diesem Kriterium sind die Flächen, die sich innerhalb der vordefinierten Entfernung von den anderen Körpern befinden.

  Mit diesen Funktionen kann man nach der Genauigkeit von unterschiedlichen CAD-Modellen den Grenzwert selbst einstellen und die entsprechenden Flächen herausfinden. Gleichzeitig sollen die Nachteile des Algorithmus nicht übersehen werden. Einerseits, haben die Informationen zur Flächenpaaren und Kollisionstypen auf diese Weise verloren. Andererseits, gehören dazu viele nicht verwandte Flächen, von denen sich nur eine Kante oder ein Punkt im Mindestabstand befindet und keine Informationen für den Einschubprozess beinhalten.

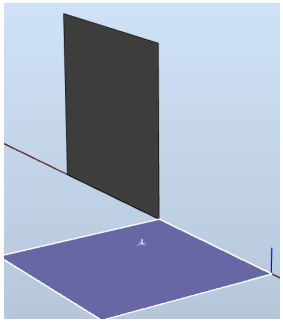


Abbildung 25: Nur ein Punkt von Flächen innerhalb des Abstandes

- **Die Kombination von zwei Lösungen**
  Die beiden Algorithmen haben ihre eigenen Vorteile und Nachteile, deshalb habe ich eine Kombination von diesen zwei Algorithmen versucht.

  - **Erster Schritt:** Mit der Entfernungsberechnung werden alle Flächen von Festkörper und Einschubkörper getrennt innerhalb des vordefinierten Mindestabstandes ausgewählt
  - **Zweiter Schritt:** Lassen sich die Flächen verbleiben, die in einem anderen Körper eine entsprechende Fläche finden können, die sich innerhalb des Abstandes befindet und deren Normalvektor in entgegengesetzte Richtung weisen.

  Durch die Kombination von zwei Lösungen reduziert die Anzahl der ausgewählten Flächen dramatisch und die nicht relevante Flächen werden ausgeschlossen.

Extrahierung der möglichen Einschubrichtungen Um die potenziellen Einschubrichtungen von den ausgewählten Flächen extrahieren zu können, muss man zuerst festlegen, welche geometrische Informationen von einer Fläche der möglichen Einschubrichtung entsprechen. Nach der Analyse von unterschiedlichen Situationen kann ich annehmen, dass in den meisten Fällen die Einschubrichtung immer identisch mit einer der folgenden Richtungen ist:

- Die Richtung einer Kante auf einer Seitenfläche
- Die Richtung der Normalvektor auf Unterseite, falls es existiert
- Das Kreuzprodukt zweier Normalenvektoren, die nicht gegeneinader sind, auf zwei verschiedenen Seitenflächen.

Die nächste Aufgabe ist die Einordnung der ausgewählten Flächen. Am Anfang habe ich die Flächen in zwei Gruppen eingeteilt und das Einteilungskriterium ist ob eine einseitige Verlängerung des Normalvektors von der Flächemitte den Körper berührt. Aus der Gruppe, die den Körper nicht berührt, können wir die Richtung nach außen durch Summation ihrer Normalenvektoren bestimmen. Danach habe ich alle folgende Analyse in den anderen Gruppen durchgeführt, weil die Seitenflächen, deren Kante der Einschubrichtung entsprechen, befinden sich am meisten in der zweiten Gruppe.

Erstens, habe ich die Geometrie berücksichtigt. Wenn die dahinter liegende Geometrie einer Fläche ein Kegel ist, können wir annehmen, dass die Achse des Kegels eine der möglichen Einschubrichtungen ist. Wenn die ausgewählten Flächen in der zweiten Gruppe mindestens zwei Seitenflächen enthalten, die sich nicht gegenüberliegen, können wir davon ausgehen, dass das Kreuzprodukt

ihrer Normalenvektoren eine der möglichen Einschubrichtungen ist. Außerdem wenn die einseitige Verlängerung einer Kante in der Seitenfläche den Körper nicht berührt, können wir auch annehmen, dass die Richtung der Kante eine der möglichen Einschubrichtungen ist. Alle mögliche Richtungen werden gespeichert und für den näschsten Schritt verwendet.

Festlegung der potentialsten Einschubrichtung Nach der Suche nach der möglichen Einschubrichtungen muss ich auch entweder die Anzahl der Einschubrichtungen einschränken oder die möglichste Einschubrichtung herausfinden. Eine intuitive Lösung dafür ist „Entity-Entity-Clash“. Die Grundidee besteht darin, wenn der Einschubkörper entlang der Einschubrichtung herausgezogen werden kann, muss es in jede beliebige Position keine Kollision gibt.

Das Problem ist wie kann man den Abstand der Schiebung des Einschubkörpers für den Kollisionstest festlegen. Egal der Abstand zu groß oder zu klein ist, wird der Test fehlgeschlagen, d.h. keine Kollision existiert aber falsche Richtung. Die endliche Lösung läutert wie folgendes:

- Aus den endgültig ausgewählten Flächen im Fest- und Einschubkörper kann ich zwei Begrenzungsrahmen bestimmen und lassen sich die zwei Begrenzungsrahmen schneiden, um einen kleineren Begrenzungsrahmen zu erhalten.
- Festlegung der Bewegungsdistanz, indem die Entfernung entlang der Richtung innerhalb des Begrenzungsrahmens berechnen und mit einem Faktor wie 0.5 multiplizieren.

Wenn entlang der Richtung keine Kollision gibt, kann die Richtung als einer der möglichsten Einschubrichtungen betrachtet wird.

Berechnung der Einschubdistanz Die Einschubdistanz muss gewährleisten, dass bei Anfangsposition von Einschubprozess keine Kollision zwischen beiden Körpern existiert. Von diesem Gedanken kann ich ausgehen, dass die Minimumeinschubdistanz zwischen Festkörper und Einschubkörper ist die Subtraktion von Maximaltiefe in Einschubkörper und Minimaltiefe in Festkörper entlang Einschubrichtung. Der Algorithmus für die Berechnung ist in diesem Fall ziemlich eindeutig und einfach.

- Finden alle Eckpunkte in ausgewählten Flächen in beiden Körpern
- Projektieren alle Eckpunkte in die Einschubrichtung und Finden die Maximaltiefe für Einschubkörper und Minimaltiefe für Festkörper
- $Einschubdistanz = Maximaltiefe - Minimaltiefe$

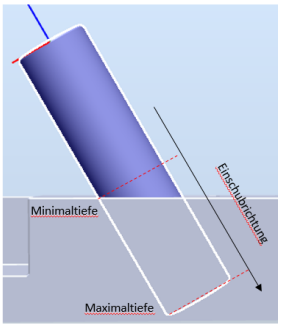


Abbildung 26: Ein Beispiel für die Berechung der Einschubdistanz

4 Ergebnis und Diskussion

Geltungsbereich Nach der Implementierung von dem obengenannten Algorithmus in unterschiedlichen CAD-Modellen, kann ich den Geltungsbereich feststellen.

- Die Einschubrichtung muss einzigartig sein oder sich in zwei entgegengesetzten Richtungen einer Linie befindet.
- Der einschub-relevante Bereich in Festkörper muss geschlossen sein.

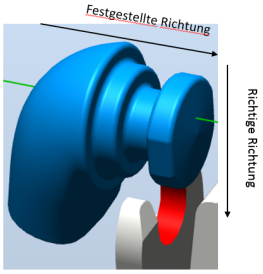


Abbildung 27: Ein Beispiel für nicht geschlossenen einschub-relevanten Bereich

  Wie obere Abbildung gezeigt, die Einschubrichtung wird normalerweise auf diesem Fall durch die Achse von Kegel- oder Zylinderoberfläche festgelegt. Aber die richtige Richtung ist anders.

- Das Profil des Einschubkörpers und des Festkörpers im einschub-relevanten Bereich kann unter den meisten Umständen übereinstimmen.
  Wie untere Abbildung gezeigt, die tatsächlich Berührungsbereich von beiden Körpern besteht nur aus Kanten und keine Flächen.
- Die beiden Körpern müssen als starrer Körper betrachtet werden, d.h. beim Einschubprozess es keine Deformation gibt und das Einrasten ist auch nicht erlaubt.

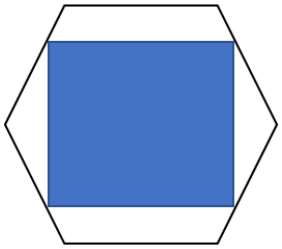
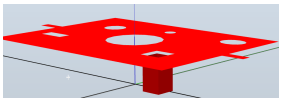


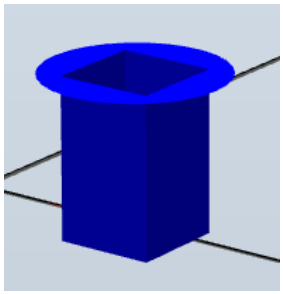
Abbildung 28: Ein Schnittansicht von einschub-relevanten Bereich

Abbildung 29: Ein Beispiel für das Einrasten

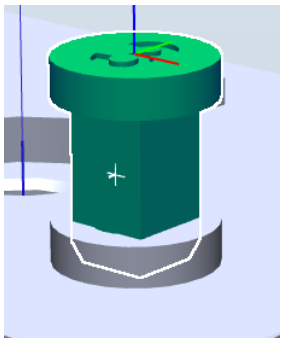
Ergebnisse für einige CAD-Modelle



(a) Die gefundene Flächen in Festkörper

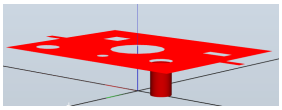


(b) Die gefundene Flächen in Einschubkörper

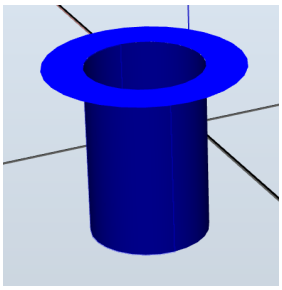


(c) Die gefundene Endposition

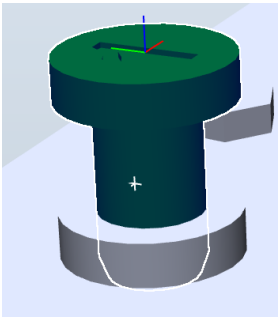
Abbildung 30: Das erste Ergebnis für Festlegung des Einschubprozesses



(a) Die gefundene Flächen in Festkörper

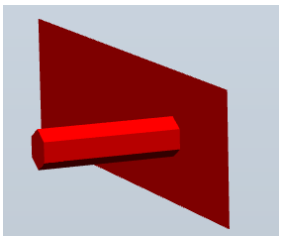


(b) Die gefundene Flächen in Einschubkörper

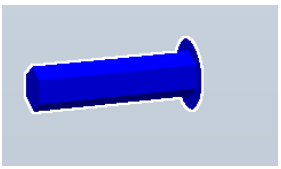


(c) Die gefundene Endposition

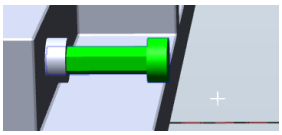
Abbildung 31: Das zweite Ergebnis für Festlegung des Einschubprozesses



(a) Die gefundene Flächen in Festkörper

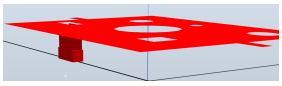


(b) Die gefundene Flächen in Einschubkörper

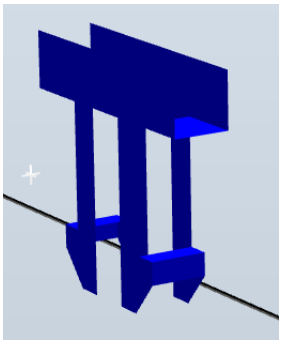


(c) Die gefundene Endposition

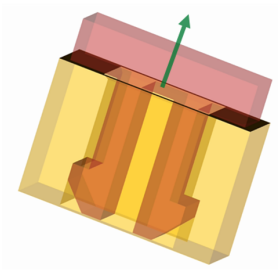
Abbildung 32: Das dritte Ergebnis für Festlegung des Einschubprozesses



(a) Die gefundene Flächen in Festkörper



(b) Die gefundene Flächen in Einschubkörper



(c) Keine gefundene Endposition wegen Einrasten

Abbildung 33: Das vierte Ergebnis für Festlegung des Einschubprozesses

5 Zusammenfassung

In den meisten Fällen kann mithilfe des Algorithmus die erwünschte Einschubrichtung und -distanz festlegen und richtige Informationen für den nächsten Schritt

nämlich die automatische Generierung von Roboterprogrammierung erfassen. Wegen der Komplexität von Einschubarten und der Toleranzunterschied von CAD-Modellen kann der Algorithmus auf einige Fälle nicht die richtige Einschubrichtung finden. Grundsätzlich handelt es sich um zwei Probleme. Erstens, wie man die Einschubrichtung in gewählten Flächen extrahieren soll, sodass die richtige Richtung nicht übersehen werden kann. Zweitens, die Überprüfung der gerechneten Richtungen kann auf einige Fälle auch nicht funktionieren wegen der Toleranz der CAD-Modellen.

Durch meine Praktikumsarbeit habe ich die Geometrie und Topologie in Computer Grafik besser verstanden und bei Implementierung des Algorithmus habe ich auch RobotStudio und ACIS API vertraut gemacht. Die automtaische Festlegung des Einschubprozesses kann als Ansatz für weitere ähnliche Arbeit für die Automatisierung des Montageprozesses betrachtet werden.

Literatur

[1] wikipedia:Off-line programming (robotics)

[2] ABB:Offline-Programmierung leicht gemacht!

[3] wikipedia:geometrie

[4] wikipedia:Geodaten

[5] wikipedia: boundary representation

[6] wikipedia: Kurve (Mathematik)

[7] wikipedia: C++/CLI

Abbildungsverzeichnis

Abkürzungsverzeichnis

OLP Offline-Programmierung 2

TCP Werkzeugarbeitspunkt 4

SDK Software Development Kit 5

b-rep boundary representation 6

STEP the Standard for the Exchange of Product Model data 6

GUI grafische Benutzeroberfläche 15

ACIS 3D ACIS Modeler 20

API application programm interface 14

CLI common language infrastructure 21

CIL common intermediate language 21

GC garbage collection 21

Codeabschnitte

- Codeabschnitte in C#

```
private void CreateBtn_Click(object sender, EventArgs e)
{
   //Start a undo proecess
   Project.UndoContext.BeginUndoStep("Direction Detection");
   try
   {
      Station station = Station.ActiveStation;
      //check if it exists an active station
      if (station == null)
      {
         Logger.AddMessage("No open Document");
         return;
      }

      //get the user selected object
      object ob1 = this.HostBodyTB.Tag;
      object ob2 = this.InsertBodyTB.Tag;
      if (ob1 is Body && ob2 is Body)
      {
         Body baseBody = (Body)ob1;
         Body insertBody = (Body)ob2;
         //copy the selected base and insert body
         Body copyBaseBody = baseBody.Copy() as Body;
         Body copyInsertBody = insertBody.Copy() as Body;
         copyBaseBody.Transform.GlobalMatrix = baseBody.Transform.GlobalMatrix;
         copyInsertBody.Transform.GlobalMatrix = insertBody.Transform.GlobalMatrix;
         //get the inside acis entity
         IntPtr acisBody1 = GetAcisEntity(copyBaseBody);
         IntPtr acisBody2 = GetAcisEntity(copyInsertBody);
         //define the index of face and direction in order to show in the
         user interface in robotstudio
         HashSet<Tuple<int, int, int>> faceIndexes = new HashSet<Tuple<int, int, int>>();
         HashSet<Tuple<int, int>> closeFaceIndexes = new HashSet<Tuple<int, int>>();
         HashSet<Tuple<int, int>> selectedIndexes = new HashSet<Tuple<int, int>>();
         HashSet<Tuple<Vector3,double>> directions = new HashSet<Tuple<Vector3,double>>();
         Utils.Instance.CreateInsertDirection(acisBody1, acisBody2, ref faceIndexes,
         ref closeFaceIndexes, ref directions);
         if(directions.Count == 0)
         {
            Logger.AddMessage(
            new LogMessage("No valid direction has been found", LogMessageSeverity.Warning));
```

```
}
foreach (Tuple<int, int> closeFaceIndex in closeFaceIndexes)
{
   Tuple<int, int, int> selected =
   faceIndexes.
   First(t => t.Item1 == closeFaceIndex.Item1 && t.Item2 == closeFaceIndex.Item2);
   selectedIndexes.Add(Tuple.Create(selected.Item1, selected.Item3));
}
//define the base part to display the relevant faces in base body
Part basePart = new Part()
{
   Name = "BaseFaces"
};
//define the insert part to display the relevant faces in insert body
Part insertPart = new Part()
{
   Name = "InsertFaces"
};
//define the move part to display the insert initial position of the insert body
Part movedPart = new Part()
{
   Name = "Moved Insert Bodies"
};
foreach (Tuple<int, int> index in selectedIndexes)
{
   //add the relevant faces from base body and insert body
   if (index.Item1 == 0)
   {
      Face f = baseBody.Faces[index.Item2];
      Body b = Body.CreateFromFace(f);
      b.Transform.GlobalMatrix = baseBody.Transform.GlobalMatrix;
      b.Color = Color.Red;
      basePart.Bodies.Add(b);
   }
   else
   {
      Face f = insertBody.Faces[index.Item2];
      Body b = Body.CreateFromFace(f);
      b.Transform.GlobalMatrix = insertBody.Transform.GlobalMatrix;
      b.Color = Color.Blue;
      insertPart.Bodies.Add(b);
   }
}
//add the possible initial positions of insert body
foreach(Tuple<Vector3,double> t in directions)
```

```
            {
               Vector3 dir = t.Item1;
               double dist = t.Item2;
               Body movedInsert = insertBody.Copy() as Body;
               movedInsert.Color = Color.Green;
               Matrix4 insertTransf = insertBody.Transform.GlobalMatrix;
               insertTransf.Translate(dist * dir);
               movedInsert.Transform.GlobalMatrix = insertTransf;
               movedPart.Bodies.Add(movedInsert);
            }
            //create a component group to contain all the faces and initial positions
            GraphicComponentGroup group = new GraphicComponentGroup()
            {
            Name = "ResultDemo"
            };
            group.GraphicComponents.Add(basePart);
            group.GraphicComponents.Add(insertPart);
            group.GraphicComponents.Add(movedPart);
            station.GraphicComponents.Add(group);
         }
      }
      catch (Exception ex)
      {
         Logger.AddMessage(ex.ToString(), LogMessageSeverity.Error);
         Project.UndoContext.CancelUndoStep(CancelUndoStepType.Rollback);
      }
      finally
      {
         this.CancelBtn_Click(sender, e);
         Project.UndoContext.EndUndoStep();
      }
   }

   private void SelectedObjects_ObjectAdded(object sender, SelectionEventArgs e)
   {
      //if the window is inactive do nothing
      if (!this.toolWindow.Visible || !this.toolWindow.DefaultTabVisibility) return;
      //add the select body
      if(e.SelectionObject is Body b)
      {
         if (this.selectInsert)
         {
            this.InsertBodyTB.Tag = b;
            this.InsertBodyTB.Text = b.Parent.DisplayName + "-" + b.DisplayName;
            this.selectInsert = false;
```

```
            }
            else
            {
                this.HostBodyTB.Tag = b;
                this.HostBodyTB.Text = b.Parent.DisplayName + "-" + b.DisplayName;
                this.selectInsert = true;
            }
        }

    }
    //get the acis entity inside the body
    private static IntPtr GetAcisEntity(Body b)
    {
        Part part = b.Parent as Part;
        GeoPart geoPart = StationsHelper.GetGeoPart(part);
        IntPtr acisEntity = new IntPtr();
        foreach (BodyDefinition bd in geoPart.BodyDefinitions)
        {
            if (bd.GetBody(part).UniqueId.Equals(b.UniqueId))
            {
                acisEntity = bd.AcisEntity;
                break;
            }
        }
        return acisEntity;
    }
```

- Codeabschnitte in C++/CLI

```
    void Utils::CreateInsertDirection(IntPtr ptrBase, IntPtr ptrInsert,
     [In, Out]HashSet<Tuple<int,int,int>^>^% faceIndexes,
     [In, Out]HashSet<Tuple<int, int>^>^% closeFaceIndexes,
     [In, Out]HashSet<Tuple<Vector3, double>^>^%noTouchDirections) {
    //get the base body and insert body
    void* acisBase = (void*)ptrBase;
    BODY* base = (BODY*)acisBase;
    void* acisInsert = (void*)ptrInsert;
    BODY* insert = (BODY*)acisInsert;
    //propogate the transformation into the geometry
    CheckOutcome(api_change_body_trans(base, NULL));
    CheckOutcome(api_change_body_trans(insert, NULL));
    API_BEGIN
    {
        EXCEPTION_BEGIN
```

```
EXCEPTION_TRY
{
//get the faces in base body and insert body, which are closed to each other
ENTITY_LIST baseFaces = GetClosestFaceList(base, insert, 0.001,0,faceIndexes);
ENTITY_LIST insertFaces = GetClosestFaceList(insert, base, 0.001,1,faceIndexes);
baseFaces.init();
insertFaces.init();

//get the faces in the close faces, whose normal vectors are opposite to each other
ENTITY_LIST closestBaseFaces;
ENTITY_LIST closestInsertFaces;
eed_output_handle* eoh = NULL;
int baseIndex = 0, insertIndex = 0;
while (FACE* baseFace = (FACE*)baseFaces.next()) {
   insertFaces.init();
   insertIndex = 0;
   while (FACE* insertFace = (FACE*)insertFaces.next()) {
      CheckOutcome(api_entity_entity_distance(baseFace, insertFace, eoh));
      const eed_output_query eoq(eoh);
      const double dist = eoq.get_distance();
      if (dist <= 0.001) {
         const SPAposition basePos = eoq.get_position(0);
         const SPAposition insertPos = eoq.get_position(1);
         SPAunit_vector baseNormal = sg_get_face_normal(baseFace, basePos);
         SPAunit_vector insertNormal = sg_get_face_normal(insertFace, insertPos);
         if (antiparallel(baseNormal, insertNormal)) {
            closestBaseFaces.add(baseFace);
            closestInsertFaces.add(insertFace);
            closeFaceIndexes->Add(Tuple::Create(0, baseIndex));
            closeFaceIndexes->Add(Tuple::Create(1, insertIndex));
         }
      }
      insertIndex++;
   }
   baseIndex++;
}
ACIS_DELETE eoh;
//initialize the list
closestBaseFaces.init();
closestInsertFaces.init();
//find the bounding box
SPAposition min_pt_base;
SPAposition max_pt_base;
SPAposition min_pt_insert;
```

```
SPAposition max_pt_insert;
SPAboxing_options* box_opts = ACIS_NEW SPAboxing_options();
api_get_entity_box(closestBaseFaces, min_pt_base, max_pt_base, box_opts);
api_get_entity_box(closestInsertFaces, min_pt_insert, max_pt_insert, box_opts);
ACIS_DELETE box_opts;
//intersecting two bounding boxes
SPAposition min_pt = MaxPos(min_pt_insert, min_pt_base);
SPAposition max_pt = MinPos(max_pt_insert, max_pt_base);
//find the potential directions by faces
HashSet<Vector3>^ directions =
FindInsertDirectionByFaces(closestBaseFaces, base, insert);
if (directions->Count == 0) {
return;
}
else {
Logger::AddMessage(String::Format("Total {0} potential directions have
 been found",directions->Count));
}
//check the collision to determine the most potential insert directions
for each(Vector3 dir in directions) {
   SPAtransf tempTransf = GetTempTransf(min_pt, max_pt, dir);
   CheckOutcome(api_transform_entity(insert, tempTransf));

   body_clash_result result;
   CheckOutcome(api_clash_bodies(insert, base, result));
   body_clash_type clashType = result.clash_type();

   if (clashType == CLASH_ABUTS || clashType == CLASH_NONE) {
      double dist = FindInsertDistance(closestBaseFaces,
      closestInsertFaces,
      SPAunit_vector(-dir.x, -dir.y, -dir.z));
      noTouchDirections->Add(Tuple::Create(dir,dist));
   }
   CheckOutcome(api_transform_entity(insert, tempTransf.inverse()));
   //reverse the translation
   SPAtransf reverse_tempTransf = translate_transf(-(tempTransf.translation()));
   CheckOutcome(api_transform_entity(insert, reverse_tempTransf));
   CheckOutcome(api_clash_bodies(insert, base, result));
   clashType = result.clash_type();
   if (clashType == CLASH_ABUTS || clashType == CLASH_NONE) {
      double dist = FindInsertDistance(closestBaseFaces, closestInsertFaces,
      SPAunit_vector(dir.x, dir.y, dir.z));
      noTouchDirections->Add(Tuple::Create(-dir,dist));
   }
   CheckOutcome(api_transform_entity(insert, reverse_tempTransf.inverse()));
```

```
   }
   }
   EXCEPTION_CATCH_FALSE
   String^ smsg = String::Format("*** ACIS Error: {0}",
   gcnew String(find_err_mess(result.error_number())));
   LogMessage^ msg = gcnew LogMessage(smsg,
   ABB::Robotics::RobotStudio::LogMessageSeverity::Error);
   Logger::AddMessage(msg);
   EXCEPTION_END
   }
API_END
}
//find the insert direction by plane faces
HashSet<Vector3>^ Utils::FindInsertDirectionByFaces(ENTITY_LIST base_face_list,
BODY * ptrBase, BODY * ptrInsert)
{
HashSet<Vector3>^ directions = gcnew HashSet<Vector3>();
SPAvector outside(0, 0, 0);
SPAunit_vector direction(0, 0, 0);
API_BEGIN
{
   EXCEPTION_BEGIN
   EXCEPTION_TRY
   {
   base_face_list.init();
   ENTITY_LIST base_not_touch_faces;
   ENTITY_LIST base_touch_faces;
   int num_not_touch = 0;
   int num_touch = 0;
   int num_cone = 0;
   while (FACE* face = (FACE*)base_face_list.next()) {
      SURFACE* surface = face->geometry();
      if (is_CONE(surface)) {
         FindDirectionByConeFace(face, ptrInsert, directions);
         num_cone++;
      }
      else if (is_PLANE(surface)) {
         if (!IsFaceNormalExtensionTouch(face, ptrBase)) {
            base_not_touch_faces.add(face);
            num_not_touch++;
         }
         else {
            base_touch_faces.add(face);
            num_touch++;
            }
```

```
   }
}

//if there are more than one face, which normal vector extension
//doesn't touch the base, we can determine the direction of /"outside/"
if (base_not_touch_faces.count() >= 1) {
   outside = GetNormalVectorSum(base_not_touch_faces);
   if (outside.len() > SPAresabs * 100){
      SPAunit_vector outsideUnit = normalise(outside);
      RoundDirection(outsideUnit);
      directions->Add(Vector3(outsideUnit.x(), outsideUnit.y(), outsideUnit.z()));
   }
}
//if the touch faces are more than twice, the insert direction can be
//found by cross product from two normal vectors
if (base_touch_faces.count() >= 2) {
   base_touch_faces.init();
   int count = directions->Count;
   FindInsertDirectionByCrossProduct(base_touch_faces,directions);
   ENTITY_LIST not_bottom_faces = FindNotBottomFaces(base_touch_faces);
   not_bottom_faces.init();
   if (count == directions->Count) {
      direction =
      normalise(FindInsertDirectionByEdge(not_bottom_faces, ptrBase, outside));
      RoundDirection(direction);
      if (!direction.is_zero()) {
         directions->Add(Vector3(direction.x(), direction.y(), direction.z()));
      }
   }
}
//if the touch face is only one, the insert direction can be
//found by one of the edge in this face
else if (base_touch_faces.count() == 1) {
   base_touch_faces.init();
   FACE* face = (FACE*)base_touch_faces.first();
   if (!IsBottom(face)) {
      direction = normalise(FindInsertDirectionByEdge(face, ptrBase, outside));
      RoundDirection(direction);
      if (!direction.is_zero()) {
         directions->Add(Vector3(direction.x(), direction.y(), direction.z()));
      }
   }
   else {
      direction = GetFaceNormalVector(face);
      RoundDirection(direction);
```

```
            if (!direction.is_zero()) {
               directions->Add(Vector3(direction.x(), direction.y(), direction.z()));
            }
         }
      }
      //if no valiable face has been found, the insert direction can be
      //found by the normal vector of the bottom face
      else {
         FACE* bottom_face = FindBottomFace(base_not_touch_faces);
         if (bottom_face != NULL) {
            direction = GetFaceNormalVector(bottom_face);
            RoundDirection(direction);
            if (!direction.is_zero()) {
               directions->Add(Vector3(direction.x(), direction.y(), direction.z()));
            }
         }
      }
   }
   EXCEPTION_CATCH_FALSE
   String^ smsg =
   String::Format("*** ACIS Error: {0}", gcnew String(find_err_mess(result.error_number())));
   LogMessage^ msg =
   gcnew LogMessage(smsg, ABB::Robotics::RobotStudio::LogMessageSeverity::Error);
   Logger::AddMessage(msg);
   EXCEPTION_END
   }
   API_END

   return directions;
   }
```