幾何座標系統與 Tag 對齊設計文件

專案名稱: IDTF Fast Prototype - Geometry & Tag Alignment
版本: 1.0
作者: Michael Lin 林志錚
組織: HTFA/Digital Twins Alliance
日期: 2025-10-22

執行摘要

本文件針對 IDTF 快速原型專案中的幾何座標系統和 Tag 位置對齊問題，提供完整的技術解決方案。

核心問題： - Tag 位置目前僅以「相對資產原點 + 平移」處理，未考慮層級變換、旋轉、縮放與單位制 - FBX → USD 轉換流程未保證法向、索引、三角化與單位一致性 - 缺乏完整的 Transform 矩陣鏈處理

解決方案： - 在 IADL 中明確規定座標系統和單位制 - 標準化 FBX → USD 轉換流程 - 實作完整的 Transform 矩陣鏈計算 - 提供 Golden Cases 單元測試驗證

目錄

---

1. 問題識別與分析

1.1 當前實作的問題

問題 1：簡化的 Tag 位置處理

當前實作：

```
# 錯誤的簡化實作
tag_world_position = asset_position + tag_local_offset
```

問題： - 未考慮父節點的變換 - 未考慮資產的旋轉和縮放 - 未考慮層級嵌套的變換累積

影響： - Tag 位置在旋轉或縮放資產後不正確 - 嵌套資產（如機器人手臂的多個關節）的 Tag 位置錯誤 - 無法正確處理複雜的場景層級

問題 2：FBX → USD 轉換不一致

當前問題： - 座標系統可能不一致（Y-up vs Z-up） - 單位制可能不一致（米 vs 毫米） - 法向量可能翻轉 - 三角化方式不統一

影響： - 模型在不同工具中顯示不一致 - Tag 位置在不同座標系統中錯誤 - 碰撞檢測失效

問題 3：缺乏驗證機制

當前問題： - 沒有標準測試案例 - 沒有誤差驗證 - 沒有回歸測試

影響： - 難以發現和修復 Bug - 難以保證品質 - 難以進行版本升級

1.2 正確的 Transform 矩陣鏈

正確的計算方式：

```
Tag 世界座標 = RootXform × Parent1Xform × ... × ParentNXform × AssetXform × TagLocalXform
```

每個 Xform 包含： - Translation（平移） - Rotation（旋轉） - Scale（縮放）

範例：

```
機器人手臂 Tag 位置計算：
World = BaseXform × Shoulder Joint Xform × Elbow Joint Xform × Wrist Xform ×
Sensor Tag Local
```

---

2. 統一座標系與單位制規範

2.1 IADL 座標系統規範

在 IADL 中明確定義座標系統和單位制：

```yaml
# IADL Asset Definition
asset:
  asset_id: "asset_001"
  name: "Industrial Robot"

  # 座標系統與單位制（必填）
  coordinate_system:
    length_unit: "m"  # 長度單位：m（米）、mm（毫米）、cm（厘米）
    up_axis: "Z"  # 向上軸：Y 或 Z
    handedness: "right"  # 手性：right（右手系）或 left（左手系）
    front_axis: "Y"  # 前方軸：X、Y 或 Z

  # 3D 模型
  model:
    usd_file: "robot.usd"
    # USD 檔案必須遵循相同的座標系統規範

  # IOT Tags
  tags:
    - tag_id: "temp_sensor_01"
      name: "Temperature Sensor"
      # Tag 位置使用相對於資產原點的局部座標
      # 單位與 length_unit 一致
      local transform:
        translation: [0.5, 0.0, 1.2]  # [x, y, z] in meters
        rotation: [0.0, 0.0, 0.0]  # [rx, ry, rz] in degrees
        scale: [1.0, 1.0, 1.0]  # [sx, sy, sz]
```

2.2 座標系統轉換規則

標準座標系統：

屬性	標準值	說明
length_unit	`m`（米）	所有 IDTF 內部計算使用米為單位
up_axis	`Z`	Z 軸向上（符合 USD 預設）
handedness	`right`	右手座標系統
front_axis	`Y`	Y 軸向前

轉換矩陣：

```
 # 從 Y-up 轉換到 Z-up
def y_up_to_z_up_matrix():
    """
    Y-up (e.g., FBX) → Z-up (USD standard)
    Rotate -90° around X-axis
    """
    return np.array([
        [1,  0,  0,  0],
        [0,  0, -1,  0],
        [0,  1,  0,  0],
        [0,  0,  0,  1]
    ])

# 單位轉換
def convert_length_unit(value, from_unit, to_unit):
    """
    轉換長度單位
    """
    units = {
        'mm': 0.001,
        'cm': 0.01,
        'm': 1.0,
        'in': 0.0254,
        'ft': 0.3048
    }

    return value * (units[from_unit] / units[to_unit])
```

2.3 USD Stage Metadata 設定

在 USD 檔案中固化座標系統資訊：

```python
# 設定 USD Stage Metadata
from pxr import Usd, UsdGeom

def setup_usd_stage_metadata(stage: Usd.Stage):
    """設定 USD Stage 的座標系統 Metadata"""

    # 設定 Up Axis
    UsdGeom.SetStageUpAxis(stage, UsdGeom.Tokens.z)

    # 設定 Meters Per Unit（單位制）
    UsdGeom.SetStageMetersPerUnit(stage, 1.0)  # 1 單位 = 1 米

    # 設定自定義 Metadata
    stage.SetMetadata('customLayerData', {
        'idtf_coordinate_system': {
            'length_unit': 'm',
            'up_axis': 'Z',
            'handedness': 'right',
            'front_axis': 'Y'
        }
    })
```

---

3. FBX → USD 轉換標準化

3.1 轉換流程

```
FBX 檔案
    ↓
1. 載入 FBX（使用 FBX SDK 或 USD FileFormatPlugin）
    ↓
2. 檢測座標系統與單位制
    ↓
3. 轉換到標準座標系統（Z-up, 米, 右手系）
    ↓
4. 標準化幾何（三角化、法向、UV）
    ↓
5. 寫入 USD 檔案
    ↓
6. 驗證轉換結果
    ↓
USD 檔案（標準化）
```

3.2 實作：FBX → USD 轉換器

```python
# iadl_designer/services/fbx_to_usd_converter.py
from pxr import Usd, UsdGeom, Gf
import numpy as np
from typing import Dict, Any, Optional

class FBXToUSDConverter:
    """FBX 到 USD 的標準化轉換器"""

    def __init__(self):
        self.target_up_axis = 'Z'
        self.target_unit = 'm'
        self.target_handedness = 'right'

    def convert(self, fbx_path: str, usd_path: str,
                options: Optional[Dict[str, Any]] = None) -> bool:
        """
        轉換 FBX 到 USD

        Args:
            fbx_path: FBX 檔案路徑
            usd_path: 輸出 USD 檔案路徑
            options: 轉換選項

        Returns:
            success: 是否成功轉換
        """
        try:
            # 1. 使用 USD 的 FBX 插件載入（如果可用）
            # 或使用 FBX SDK 手動解析
            stage = Usd.Stage.Open(fbx_path)

            if not stage:
                # Fallback: 使用 FBX SDK
                stage = self._load_fbx_with_sdk(fbx_path)

            # 2. 檢測原始座標系統
            source_metadata = self._detect_coordinate_system(stage)

            # 3. 轉換到標準座標系統
            self._convert_coordinate_system(stage, source_metadata)

            # 4. 標準化幾何
            self._standardize_geometry(stage, options)

            # 5. 設定 Stage Metadata
            setup_usd_stage_metadata(stage)

            # 6. 儲存 USD 檔案
            stage.Export(usd_path)

            # 7. 驗證轉換結果
            if not self._validate_conversion(usd_path):
                print("[FBX→USD] Warning: Conversion validation failed")

            print(f"[FBX→USD] Successfully converted: {fbx_path} → {usd_path}")
            return True

        except Exception as e:
            print(f"[FBX→USD] Error converting FBX to USD: {e}")
            return False

    def _detect_coordinate_system(self, stage: Usd.Stage) -> Dict[str, Any]:
```

```
        """檢測原始座標系統"""
        metadata = {}

        # 檢測 Up Axis
        up_axis_token = UsdGeom.GetStageUpAxis(stage)
        metadata['up_axis'] = str(up_axis_token).upper()  # 'Y' or 'Z'

        # 檢測單位制
        meters_per_unit = UsdGeom.GetStageMetersPerUnit(stage)
        if meters_per_unit == 1.0:
            metadata['length_unit'] = 'm'
        elif meters_per_unit == 0.01:
            metadata['length_unit'] = 'cm'
        elif meters_per_unit == 0.001:
            metadata['length_unit'] = 'mm'
        else:
            metadata['length_unit'] = 'm'  # Default
            print(f"[FBX→USD] Unknown unit scale: {meters_per_unit}, assuming
meters")

        # 檢測手性（通常 FBX 是右手系）
        metadata['handedness'] = 'right'

        return metadata

    def _convert_coordinate_system(self, stage: Usd.Stage, source_metadata:
Dict[str, Any]):
        """轉換座標系統"""

        # 計算轉換矩陣
        transform_matrix = Gf.Matrix4d(1.0)  # Identity

        # Up Axis 轉換
        if source_metadata['up_axis'] != self.target_up_axis:
            if source_metadata['up_axis'] == 'Y' and self.target_up_axis ==
'Z':
                # Y-up → Z-up: Rotate -90° around X-axis
                transform_matrix = Gf.Matrix4d(
                    1,  0,  0,  0,
                    0,  0, -1,  0,
                    0,  1,  0,  0,
                    0,  0,  0,  1
                )

        # 單位轉換
        if source_metadata['length_unit'] != self.target_unit:
            scale_factor = convert_length_unit(1.0,
source_metadata['length_unit'], self.target_unit)
            scale_matrix = Gf.Matrix4d(1.0)
            scale_matrix.SetScale(Gf.Vec3d(scale_factor, scale_factor,
scale_factor))
            transform_matrix = transform_matrix * scale_matrix

        # 應用轉換到所有 Prims
        if not transform_matrix.IsIdentity():
            self._apply_transform_to_stage(stage, transform_matrix)

    def  apply_transform_to_stage(self, stage: Usd.Stage, transform_matrix:
Gf.Matrix4d):
        """應用轉換矩陣到 Stage 的所有根 Prims"""

        for prim in stage.GetPseudoRoot().GetChildren():
            if prim.IsA(UsdGeom.Xformable):
                xformable = UsdGeom.Xformable(prim)

                # 獲取當前變換
```

```
                current_xform = xformable.GetLocalTransformation()

                # 應用轉換
                new_xform = transform_matrix * current_xform

                # 設定新變換
                xformable.ClearXformOpOrder()
                xform_op = xformable.AddTransformOp()
                xform_op.Set(new_xform)

    def _standardize_geometry(self, stage: Usd.Stage, options:
Optional[Dict[str, Any]]):
        """標準化幾何"""

        options = options or {}

        for prim in stage.Traverse():
            if prim.IsA(UsdGeom.Mesh):
                mesh = UsdGeom.Mesh(prim)

                # 1. 三角化（如果需要）
                if options.get('triangulate', True):
                    self._triangulate_mesh(mesh)

                # 2. 計算法向（如果缺失）
                if options.get('compute_normals', True):
                    self._compute_normals(mesh)

                # 3. 標準化 UV（如果需要）
                if options.get('standardize_uvs', True):
                    self._standardize_uvs(mesh)

    def _triangulate_mesh(self, mesh: UsdGeom.Mesh):
        """三角化網格"""
        # 實作三角化邏輯
        # 將所有多邊形轉換為三角形
        pass

    def  compute normals(self, mesh: UsdGeom.Mesh):
        """計算法向量"""
        normals_attr = mesh.GetNormalsAttr()

        if not normals_attr or not normals_attr.Get():
            # 計算法向量
            # 這裡需要實作法向量計算邏輯
            pass

    def _standardize_uvs(self, mesh: UsdGeom.Mesh):
        """標準化 UV 座標"""
        # 確保 UV 在 [0, 1] 範圍內
        pass

    def  validate conversion(self, usd_path: str) -> bool:
        """驗證轉換結果"""
        try:
            stage = Usd.Stage.Open(usd_path)

            # 檢查 Up Axis
            up_axis = UsdGeom.GetStageUpAxis(stage)
            if str(up_axis).upper() != self.target_up_axis:
                print(f"[FBX→USD] Validation failed: Up axis is {up_axis},
expected {self.target_up_axis}")
                return False

            # 檢查單位制
            meters_per_unit = UsdGeom.GetStageMetersPerUnit(stage)
```

```
            if abs(meters_per_unit - 1.0) > 1e-6:
                print(f"[FBX→USD] Validation failed: Meters per unit is
{meters_per_unit}, expected 1.0")
                return False

            # 檢查是否有幾何
            has_geometry = False
            for prim in stage.Traverse():
                if prim.IsA(UsdGeom.Mesh):
                    has_geometry = True
                    break

            if not has_geometry:
                print("[FBX→USD] Validation failed: No geometry found in USD
file")
                return False

            return True

        except Exception as e:
            print(f"[FBX→USD] Validation error: {e}")
            return False
```

3.3 使用範例

```
# 使用 FBX → USD 轉換器
converter = FBXToUSDConverter()

# 轉換選項
options = {
    'triangulate': True,  # 三角化網格
    'compute_normals': True,  # 計算法向量
    'standardize_uvs': True  # 標準化 UV
}

# 執行轉換
success = converter.convert(
    fbx_path='models/robot.fbx',
    usd_path='models/robot.usd',
    options=options
)

if success:
    print("Conversion successful!")
```

---

4. 完整 Transform 矩陣鏈處理

4.1 Transform 矩陣鏈計算

正確的實作：

```python
# iadl_designer/utils/transform_utils.py
from pxr import Usd, UsdGeom, Gf
from typing import List, Tuple

class TransformUtils:
    """Transform 矩陣鏈工具"""

    @staticmethod
    def get_local_to_world_transform(prim: Usd.Prim, time: Usd.TimeCode =
Usd.TimeCode.Default()) -> Gf.Matrix4d:
        """
        獲取 Prim 的局部到世界座標變換矩陣

        Args:
            prim: USD Prim
            time: 時間碼

        Returns:
            transform: 4x4 變換矩陣
        """
        if not prim.IsA(UsdGeom.Xformable):
            return Gf.Matrix4d(1.0)  # Identity

        xformable = UsdGeom.Xformable(prim)

        # 使用 USD API 計算完整的變換矩陣鏈
        world_transform = xformable.ComputeLocalToWorldTransform(time)

        return world_transform

    @staticmethod
    def get_local_transform(prim: Usd.Prim, time: Usd.TimeCode =
Usd.TimeCode.Default()) -> Gf.Matrix4d:
        """
        獲取 Prim 的局部變換矩陣

        Args:
            prim: USD Prim
            time: 時間碼

        Returns:
            transform: 4x4 變換矩陣
        """
        if not prim.IsA(UsdGeom.Xformable):
            return Gf.Matrix4d(1.0)  # Identity

        xformable = UsdGeom.Xformable(prim)

        # 獲取局部變換
        local_transform = xformable.GetLocalTransformation(time)

        return local_transform

    @staticmethod
    def decompose_transform(matrix: Gf.Matrix4d) -> Tuple[Gf.Vec3d,
Gf.Rotation, Gf.Vec3d]:
        """
        分解變換矩陣為 Translation, Rotation, Scale

        Args:
            matrix: 4x4 變換矩陣

        Returns:
            (translation, rotation, scale)
        """
```

```
        # 提取平移
        translation = matrix.ExtractTranslation()

        # 提取旋轉和縮放
        rotation = matrix.ExtractRotation()

        # 計算縮放
        # 從矩陣中移除旋轉後，剩下的就是縮放
        rotation_matrix = rotation.GetQuaternion().GetMatrix()
        scale_matrix = matrix.RemoveScaleShear()

        scale_x = Gf.Vec3d(scale_matrix[0][0], scale_matrix[0][1],
scale_matrix[0][2]).GetLength()
        scale_y = Gf.Vec3d(scale_matrix[1][0], scale_matrix[1][1],
scale_matrix[1][2]).GetLength()
        scale_z = Gf.Vec3d(scale_matrix[2][0], scale_matrix[2][1],
scale_matrix[2][2]).GetLength()

        scale = Gf.Vec3d(scale_x, scale_y, scale_z)

        return (translation, rotation, scale)

    @staticmethod
    def compose_transform(translation: Gf.Vec3d, rotation: Gf.Rotation, scale:
Gf.Vec3d) -> Gf.Matrix4d:
        """
        組合 Translation, Rotation, Scale 為變換矩陣

        Args:
            translation: 平移向量
            rotation: 旋轉
            scale: 縮放向量

        Returns:
            matrix: 4x4 變換矩陣
        """
        # 創建縮放矩陣
        scale_matrix = Gf.Matrix4d(1.0)
        scale_matrix.SetScale(scale)

        # 創建旋轉矩陣
        rotation matrix = Gf.Matrix4d(1.0)
        rotation_matrix.SetRotate(rotation)

        # 創建平移矩陣
        translation_matrix = Gf.Matrix4d(1.0)
        translation_matrix.SetTranslate(translation)

        # 組合：Translation × Rotation × Scale
        transform = translation_matrix * rotation_matrix * scale_matrix

        return transform

    @staticmethod
    def transform_point(point: Gf.Vec3d, matrix: Gf.Matrix4d) -> Gf.Vec3d:
        """
        使用變換矩陣變換點

        Args:
            point: 3D 點
            matrix: 4x4 變換矩陣

        Returns:
            transformed_point: 變換後的點
        """
        # 將點轉換為齊次座標
```

```
        homogeneous_point = Gf.Vec4d(point[0], point[1], point[2], 1.0)

        # 應用變換
        transformed = matrix.Transform(homogeneous_point)

        # 轉換回 3D 座標
        return Gf.Vec3d(transformed[0], transformed[1], transformed[2])

    @staticmethod
    def get_transform_chain(prim: Usd.Prim) -> List[Gf.Matrix4d]:
        """
        獲取從根節點到指定 Prim 的完整變換鏈

        Args:
            prim: USD Prim

        Returns:
            transforms: 變換矩陣列表（從根到葉）
        """
        transforms = []

        current_prim = prim
        while current_prim and current_prim.GetPath() !=
Usd.Path.absoluteRootPath:
            if current_prim.IsA(UsdGeom.Xformable):
                local_transform =
TransformUtils.get_local_transform(current_prim)
                transforms.insert(0, local_transform)  # 插入到開頭

            current_prim = current_prim.GetParent()

        return transforms

    @staticmethod
    def compute_world_position_from_chain(transforms: List[Gf.Matrix4d],
local_point: Gf.Vec3d) -> Gf.Vec3d:
        """
        從變換鏈計算世界座標

        Args:
            transforms: 變換矩陣列表（從根到葉）
            local_point: 局部座標點

        Returns:
            world_point: 世界座標點
        """
        # 累積變換
        accumulated_transform = Gf.Matrix4d(1.0)  # Identity

        for transform in transforms:
            accumulated_transform = accumulated_transform * transform

        # 應用到點
        world_point = TransformUtils.transform_point(local_point,
accumulated_transform)

        return world_point
```

4.2 Tag 位置計算範例

```python
# 範例：計算 Tag 的世界座標
from pxr import Usd, Gf
from iadl_designer.utils.transform_utils import TransformUtils

# 開啟 USD Stage
stage = Usd.Stage.Open('robot.usd')

# 獲取資產 Prim
asset_prim = stage.GetPrimAtPath('/World/Robot/Arm')

# Tag 的局部座標（相對於資產原點）
tag_local_position = Gf.Vec3d(0.5, 0.0, 1.2)  # [x, y, z] in meters

# 方法 1：使用 USD API（推薦）
world_transform = TransformUtils.get_local_to_world_transform(asset_prim)
tag_world_position = TransformUtils.transform_point(tag_local_position,
world_transform)

print(f"Tag world position: {tag_world_position}")

# 方法 2：手動計算變換鏈（用於理解和調試）
transform_chain = TransformUtils.get_transform_chain(asset_prim)
tag_world_position_manual = TransformUtils.compute_world_position_from_chain(
    transform_chain, tag_local_position
)

print(f"Tag world position (manual): {tag_world_position_manual}")

# 驗證兩種方法的結果應該一致
assert (tag_world_position - tag_world_position_manual).GetLength() < 1e-6
```

---

5. Tag 位置編輯器改進

5.1 改進的 Tag 位置編輯器

```python
# iadl_designer/ui/tag_position_editor.py
from PyQt6.QtWidgets import (QWidget, QVBoxLayout, QHBoxLayout, QLabel,
                             QDoubleSpinBox, QPushButton, QGroupBox)
from PyQt6.QtCore import pyqtSignal
from pxr import Usd, UsdGeom, Gf
from iadl_designer.utils.transform_utils import TransformUtils

class TagPositionEditor(QWidget):
    """Tag 位置編輯器（改進版）"""

    position_changed = pyqtSignal(str, Gf.Vec3d)  # (tag_id, world_position)

    def __init__(self, stage: Usd.Stage, asset_prim: Usd.Prim):
        super().__init__()
        self.stage = stage
        self.asset_prim = asset_prim
        self.current_tag_id = None

        self.setup_ui()

    def setup_ui(self):
        """設定 UI"""
        layout = QVBoxLayout(self)

        # 局部座標組
        local_group = self.create_local_coordinate_group()
        layout.addWidget(local_group)

        # 世界座標組（只讀）
        world_group = self.create_world_coordinate_group()
        layout.addWidget(world_group)

        # 控制按鈕
        button_layout = QHBoxLayout()

        self.apply_btn = QPushButton("Apply")
        self.reset_btn = QPushButton("Reset")
        self.pick_btn = QPushButton("Pick from 3D View")

        self.apply_btn.clicked.connect(self.apply_changes)
        self.reset_btn.clicked.connect(self.reset_values)
        self.pick_btn.clicked.connect(self.pick_from_3d_view)

        button_layout.addWidget(self.apply_btn)
        button_layout.addWidget(self.reset_btn)
        button_layout.addWidget(self.pick_btn)

        layout.addLayout(button_layout)

    def create_local_coordinate_group(self) -> QGroupBox:
        """創建局部座標編輯組"""
        group = QGroupBox("Local Coordinates (relative to asset)")
        layout = QVBoxLayout()

        # Translation
        trans_layout = QHBoxLayout()
        trans_layout.addWidget(QLabel("Translation:"))
```

```
        self.local_x_spin = QDoubleSpinBox()
        self.local_y_spin = QDoubleSpinBox()
        self.local_z_spin = QDoubleSpinBox()

        for spin in [self.local_x_spin, self.local_y_spin, self.local_z_spin]:
            spin.setRange(-1000.0, 1000.0)
            spin.setDecimals(3)
            spin.setSingleStep(0.1)
            spin.valueChanged.connect(self.on_local_position_changed)

        trans_layout.addWidget(QLabel("X:"))
        trans_layout.addWidget(self.local_x_spin)
        trans_layout.addWidget(QLabel("Y:"))
        trans_layout.addWidget(self.local_y_spin)
        trans_layout.addWidget(QLabel("Z:"))
        trans_layout.addWidget(self.local_z_spin)

        layout.addLayout(trans_layout)

        group.setLayout(layout)
        return group

    def create_world_coordinate_group(self) -> QGroupBox:
        """創建世界座標顯示組（只讀）"""
        group = QGroupBox("World Coordinates (read-only)")
        layout = QVBoxLayout()

        self.world_x_label = QLabel("X: 0.000")
        self.world_y_label = QLabel("Y: 0.000")
        self.world_z_label = QLabel("Z: 0.000")

        layout.addWidget(self.world_x_label)
        layout.addWidget(self.world_y_label)
        layout.addWidget(self.world_z_label)

        group.setLayout(layout)
        return group

    def set_tag(self, tag_id: str, local_position: Gf.Vec3d):
        """設定當前編輯的 Tag"""
        self.current_tag_id = tag_id

        # 更新局部座標
        self.local_x_spin.setValue(local_position[0])
        self.local_y_spin.setValue(local_position[1])
        self.local_z_spin.setValue(local_position[2])

        # 更新世界座標
        self.update_world_coordinates()

    def on_local_position_changed(self):
        """當局部座標改變時"""
        self.update_world_coordinates()

    def update_world_coordinates(self):
        """更新世界座標顯示"""
        # 獲取局部座標
        local_position = Gf.Vec3d(
            self.local_x_spin.value(),
            self.local_y_spin.value(),
            self.local_z_spin.value()
        )

        # 計算世界座標（使用完整的變換矩陣鏈）
        world_transform =
```

```
TransformUtils.get_local_to_world_transform(self.asset_prim)
        world_position = TransformUtils.transform_point(local_position,
world_transform)

        # 更新顯示
        self.world_x_label.setText(f"X: {world_position[0]:.3f}")
        self.world_y_label.setText(f"Y: {world_position[1]:.3f}")
        self.world_z_label.setText(f"Z: {world_position[2]:.3f}")

    def apply_changes(self):
        """應用變更"""
        if not self.current_tag_id:
            return

        # 獲取局部座標
        local_position = Gf.Vec3d(
            self.local_x_spin.value(),
            self.local_y_spin.value(),
            self.local_z_spin.value()
        )

        # 計算世界座標
        world_transform =
TransformUtils.get_local_to_world_transform(self.asset_prim)
        world_position = TransformUtils.transform_point(local_position,
world_transform)

        # 發送信號
        self.position_changed.emit(self.current_tag_id, world_position)

        print(f"[TagPositionEditor] Tag {self.current_tag_id} position
updated:")
        print(f"  Local: {local_position}")
        print(f"  World: {world_position}")

    def reset_values(self):
        """重置值"""
        self.local_x_spin.setValue(0.0)
        self.local_y_spin.setValue(0.0)
        self.local_z_spin.setValue(0.0)

    def pick_from_3d_view(self):
        """從 3D 視圖中選取位置"""
        # 這裡需要與 3D 視圖整合
        # 當用戶在 3D 視圖中點擊時，獲取點擊的世界座標
        # 然後轉換為局部座標
        pass

    def world_to_local_position(self, world_position: Gf.Vec3d) -> Gf.Vec3d:
        """將世界座標轉換為局部座標"""
        # 獲取世界到局部的變換矩陣（逆矩陣）
        world_transform =
TransformUtils.get_local_to_world_transform(self.asset_prim)
        local_transform = world_transform.GetInverse()

        # 轉換點
        local_position = TransformUtils.transform_point(world_position,
local_transform)

        return local_position
```

5.2 3D 視圖整合

```python
# iadl_designer/ui/viewport_3d.py
from PyQt6.QtWidgets import QOpenGLWidget
from PyQt6.QtCore import pyqtSignal
from pxr import Gf
import numpy as np

class Viewport3D(QOpenGLWidget):
    """3D 視圖（改進版）"""

    position_picked = pyqtSignal(Gf.Vec3d)  # 當用戶點擊 3D 視圖時發送世界座標

    def __init__(self):
        super().__init__()
        self.picking_mode = False

    def enable_picking_mode(self):
        """啟用位置選取模式"""
        self.picking_mode = True
        self.setCursor(Qt.CursorShape.CrossCursor)

    def disable_picking_mode(self):
        """禁用位置選取模式"""
        self.picking_mode = False
        self.setCursor(Qt.CursorShape.ArrowCursor)

    def mousePressEvent(self, event):
        """滑鼠點擊事件"""
        if self.picking_mode:
            # 將螢幕座標轉換為世界座標
            world_position = self.screen_to_world(event.pos())

            # 發送信號
            self.position_picked.emit(world_position)

            # 禁用選取模式
            self.disable_picking_mode()
        else:
            super().mousePressEvent(event)

    def screen_to_world(self, screen_pos) -> Gf.Vec3d:
        """將螢幕座標轉換為世界座標（使用 Ray Casting）"""
        # 1. 獲取螢幕座標的 NDC（Normalized Device Coordinates）
        x = (2.0 * screen_pos.x()) / self.width() - 1.0
        y = 1.0 - (2.0 * screen_pos.y()) / self.height()

        # 2. 創建射線（從相機位置到點擊點）
        ray_origin, ray_direction = self.create_ray_from_ndc(x, y)

        # 3. 與場景進行射線投射
        hit_point = self.ray_cast(ray_origin, ray_direction)

        return hit_point

    def create_ray_from_ndc(self, ndc_x: float, ndc_y: float):
        """從 NDC 座標創建射線"""
        # 獲取相機參數
        view_matrix = self.get_view_matrix()
        projection_matrix = self.get_projection_matrix()

        # 計算逆矩陣
        inv_projection = projection_matrix.GetInverse()
        inv_view = view_matrix.GetInverse()
```

```
        # NDC → Clip Space
        clip_coords = Gf.Vec4d(ndc_x, ndc_y, -1.0, 1.0)

        # Clip Space → Eye Space
        eye_coords = inv_projection.Transform(clip_coords)
        eye_coords = Gf.Vec4d(eye_coords[0], eye_coords[1], -1.0, 0.0)

        # Eye Space → World Space
        world_coords = inv_view.Transform(eye_coords)
        ray_direction = Gf.Vec3d(world_coords[0], world_coords[1],
world_coords[2]).GetNormalized()

        # 射線原點（相機位置）
        camera_pos = inv_view.Transform(Gf.Vec4d(0, 0, 0, 1))
        ray_origin = Gf.Vec3d(camera_pos[0], camera_pos[1], camera_pos[2])

        return ray_origin, ray_direction

    def ray_cast(self, ray_origin: Gf.Vec3d, ray_direction: Gf.Vec3d) ->
Gf.Vec3d:
        """射線投射，找到與場景的交點"""
        # 這裡需要實作射線與場景幾何的交點計算
        # 可以使用 USD 的 UsdGeom.BBoxCache 或自定義的 BVH

        # 簡化實作：與 XY 平面（Z=0）的交點
        if abs(ray_direction[2]) > 1e-6:
            t = -ray_origin[2] / ray_direction[2]
            hit_point = ray_origin + ray_direction * t
            return hit_point

        return Gf.Vec3d(0, 0, 0)
```

---

6. 單元測試與驗證

6.1 Golden Cases

準備 3 套標準測試模型：

Test Case 1：簡單平移

```python
# tests/test_transform_simple.py
import unittest
from pxr import Usd, UsdGeom, Gf
from iadl_designer.utils.transform_utils import TransformUtils

class TestTransformSimple(unittest.TestCase):
    """測試簡單平移"""

    def setUp(self):
        """設定測試環境"""
        # 創建測試 Stage
        self.stage = Usd.Stage.CreateInMemory()

        # 創建資產 Prim
        asset_prim = self.stage.DefinePrim('/Asset', 'Xform')
        xformable = UsdGeom.Xformable(asset_prim)

        # 設定平移：(10, 0, 0)
        xform_op = xformable.AddTranslateOp()
        xform_op.Set(Gf.Vec3d(10, 0, 0))

        self.asset_prim = asset_prim

    def test_local_to_world_translation(self):
        """測試局部到世界座標的平移"""
        # Tag 局部座標
        tag_local = Gf.Vec3d(1, 0, 0)

        # 計算世界座標
        world_transform =
TransformUtils.get_local_to_world_transform(self.asset_prim)
        tag_world = TransformUtils.transform_point(tag_local, world_transform)

        # 預期世界座標：(11, 0, 0)
        expected = Gf.Vec3d(11, 0, 0)

        # 驗證誤差 < 1e-3
        error = (tag_world - expected).GetLength()
        self.assertLess(error, 1e-3, f"Error: {error}, Expected: {expected},
Got: {tag_world}")

if __name__ == '__main__':
    unittest.main()
```

Test Case 2：旋轉 + 平移

```python
# tests/test_transform_rotation.py
import unittest
from pxr import Usd, UsdGeom, Gf
from iadl_designer.utils.transform_utils import TransformUtils
import math

class TestTransformRotation(unittest.TestCase):
    """測試旋轉 + 平移"""

    def setUp(self):
        """設定測試環境"""
        self.stage = Usd.Stage.CreateInMemory()

        # 創建資產 Prim
        asset_prim = self.stage.DefinePrim('/Asset', 'Xform')
        xformable = UsdGeom.Xformable(asset_prim)

        # 設定平移：(10, 0, 0)
        translate_op = xformable.AddTranslateOp()
        translate_op.Set(Gf.Vec3d(10, 0, 0))

        # 設定旋轉：繞 Z 軸旋轉 90°
        rotate_op = xformable.AddRotateZOp()
        rotate_op.Set(90.0)  # degrees

        self.asset_prim = asset_prim

    def test_local_to_world_rotation(self):
        """測試局部到世界座標的旋轉"""
        # Tag 局部座標：(1, 0, 0)
        tag_local = Gf.Vec3d(1, 0, 0)

        # 計算世界座標
        world_transform =
TransformUtils.get_local_to_world_transform(self.asset_prim)
        tag_world = TransformUtils.transform_point(tag_local, world_transform)

        # 預期世界座標：
        # 1. 旋轉 90°：(1, 0, 0) → (0, 1, 0)
        # 2. 平移 (10, 0, 0)：(0, 1, 0) → (10, 1, 0)
        expected = Gf.Vec3d(10, 1, 0)

        # 驗證誤差 < 1e-3
        error = (tag_world - expected).GetLength()
        self.assertLess(error, 1e-3, f"Error: {error}, Expected: {expected},
Got: {tag_world}")

if __name__ == '__main__':
    unittest.main()
```

Test Case 3：巢狀變換（機器人手臂）

```python
# tests/test_transform_nested.py
import unittest
from pxr import Usd, UsdGeom, Gf
from iadl_designer.utils.transform_utils import TransformUtils

class TestTransformNested(unittest.TestCase):
    """測試巢狀變換（機器人手臂）"""

    def setUp(self):
        """設定測試環境"""
        self.stage = Usd.Stage.CreateInMemory()

        # 創建機器人手臂層級結構
        # Root
        #   └─ Base (translate: 0, 0, 1)
        #       └─ Shoulder (rotate Z: 45°)
        #           └─ Elbow (translate: 2, 0, 0, rotate Z: -30°)
        #               └─ Wrist (translate: 1.5, 0, 0)

        # Root
        root_prim = self.stage.DefinePrim('/Robot', 'Xform')

        # Base
        base_prim = self.stage.DefinePrim('/Robot/Base', 'Xform')
        base_xform = UsdGeom.Xformable(base_prim)
        base_xform.AddTranslateOp().Set(Gf.Vec3d(0, 0, 1))

        # Shoulder
        shoulder_prim = self.stage.DefinePrim('/Robot/Base/Shoulder', 'Xform')
        shoulder_xform = UsdGeom.Xformable(shoulder_prim)
        shoulder_xform.AddRotateZOp().Set(45.0)

        # Elbow
        elbow_prim = self.stage.DefinePrim('/Robot/Base/Shoulder/Elbow',
'Xform')
        elbow_xform = UsdGeom.Xformable(elbow_prim)
        elbow_xform.AddTranslateOp().Set(Gf.Vec3d(2, 0, 0))
        elbow_xform.AddRotateZOp().Set(-30.0)

        # Wrist
        wrist_prim = self.stage.DefinePrim('/Robot/Base/Shoulder/Elbow/Wrist',
'Xform')
        wrist_xform = UsdGeom.Xformable(wrist_prim)
        wrist_xform.AddTranslateOp().Set(Gf.Vec3d(1.5, 0, 0))

        self.wrist_prim = wrist_prim

    def test_nested_transform(self):
        """測試巢狀變換"""
        # Tag 局部座標（相對於 Wrist）
        tag_local = Gf.Vec3d(0.5, 0, 0)

        # 計算世界座標
        world_transform =
TransformUtils.get_local_to_world_transform(self.wrist_prim)
        tag_world = TransformUtils.transform_point(tag_local, world_transform)

        # 手動計算預期世界座標
        # 這裡需要手動計算完整的變換鏈
        # Base: T(0, 0, 1)
        # Shoulder: R_z(45°)
        # Elbow: T(2, 0, 0) * R_z(-30°)
        # Wrist: T(1.5, 0, 0)
```

```
        # Tag: T(0.5, 0, 0)

        # 使用數值計算或手動計算得到預期值
        # 這裡僅作為示範，實際值需要精確計算
        expected = Gf.Vec3d(3.232, 2.732, 1.0)  # 示範值

        # 驗證誤差 < 1e-3
        error = (tag_world - expected).GetLength()
        self.assertLess(error, 1e-3, f"Error: {error}, Expected: {expected},
Got: {tag_world}")

    def test_round_trip(self):
        """測試往返轉換（局部 → 世界 → 局部）"""
        # Tag 原始局部座標
        tag_local_original = Gf.Vec3d(0.5, 0.3, 0.2)

        # 局部 → 世界
        world_transform =
TransformUtils.get_local_to_world_transform(self.wrist_prim)
        tag_world = TransformUtils.transform_point(tag_local_original,
world_transform)

        # 世界 → 局部
        local_transform = world_transform.GetInverse()
        tag_local_recovered = TransformUtils.transform_point(tag_world,
local_transform)

        # 驗證往返誤差 < 1e-6
        error = (tag_local_recovered - tag_local_original).GetLength()
        self.assertLess(error, 1e-6, f"Round-trip error: {error}")

if __name__ == '__main__':
    unittest.main()
```

6.2 自動化測試腳本

```
# tests/run_all_transform_tests.py
import unittest
import sys

def run_all_tests():
    """運行所有變換測試"""

    # 創建測試套件
    loader = unittest.TestLoader()
    suite = unittest.TestSuite()

    # 添加測試
    suite.addTests(loader.loadTestsFromName('tests.test_transform_simple'))
    suite.addTests(loader.loadTestsFromName('tests.test_transform_rotation'))
    suite.addTests(loader.loadTestsFromName('tests.test_transform_nested'))

    # 運行測試
    runner = unittest.TextTestRunner(verbosity=2)
    result = runner.run(suite)

    # 返回結果
    return result.wasSuccessful()

if __name__ == '__main__':
    success = run_all_tests()
    sys.exit(0 if success else 1)
```

6.3 持續整合（CI）

```yaml
# .github/workflows/transform_tests.yml
name: Transform Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install usd-core  # USD Python bindings

    - name: Run transform tests
      run: |
        python tests/run_all_transform_tests.py

    - name: Check test coverage
      run: |
        pip install coverage
        coverage run -m pytest tests/
        coverage report --fail-under=80
```

---

7. 實作指南

7.1 實作優先順序

優先級	任務	預估時間
P0	實作 TransformUtils 工具類別	2 天
P0	更新 Tag 位置編輯器使用完整變換鏈	1 天
P0	實作 Golden Cases 單元測試	2 天
P1	實作 FBX → USD 轉換器	3 天
P1	在 IADL 中添加座標系統規範	1 天
P1	實作 3D 視圖的位置選取功能	2 天
P2	設定 CI/CD 自動化測試	1 天
P2	撰寫使用者文件	1 天

總計：約 13 天（2.5 週）

7.2 驗收標準

✅ **功能驗收**： - Tag 位置在資產旋轉、縮放後仍然正確 - 巢狀資產（如機器人手臂）的 Tag 位置正確 - 3D 視圖中可以點選位置來設定 Tag - FBX 模型轉換為 USD 後座標系統一致

✅ **品質驗收**： - 所有 Golden Cases 測試通過 - 位置誤差 < 1e-3（1 毫米） - 往返轉換誤差 < 1e-6 - 程式碼覆蓋率 > 80%

✅ **文件驗收**： - API 文件完整 - 使用者指南完整 - 測試文件完整

7.3 風險與緩解

風險	影響	機率	緩解措施
USD API 學習曲線	高	中	提前學習 USD 文件，參考範例程式碼
FBX SDK 整合困難	中	中	使用 USD 的 FBX 插件，或尋找替代方案
效能問題（大型場景）	中	低	使用快取，優化變換計算
測試案例不足	高	中	增加更多 Golden Cases，涵蓋邊界情況

總結

本文件提供了完整的幾何座標系統和 Tag 對齊解決方案，包括：

1. **統一座標系與單位制規範** - 在 IADL 中明確定義，確保一致性
2. **FBX → USD 轉換標準化** - 保證模型轉換的正確性
3. **完整 Transform 矩陣鏈處理** - 正確計算世界座標
4. **改進的 Tag 位置編輯器** - 使用 USD API 處理變換
5. **Golden Cases 單元測試** - 確保品質和可維護性

核心優勢：

✅ **正確性**：使用完整的變換矩陣鏈，支援旋轉、縮放、巢狀變換
✅ **一致性**：統一的座標系統和單位制
✅ **可測試性**：完整的單元測試和 Golden Cases
✅ **可維護性**：清晰的 API 和文件

下一步：

1. 實作 TransformUtils 工具類別
2. 更新 Tag 位置編輯器
3. 實作 Golden Cases 單元測試
4. 整合到 IADL Designer 和 FDL Designer

附錄：

- USD Transform API 文件
- FBX SDK 文件
- 座標系統轉換參考