Game Engines in AI Exercise 05

Waldemar Zeitler

December 19, 2018

Task 1

Für diese Aufgabe wurden die nötigen Klassen wie in der Aufgabe beschrieben erstellt. Dazu wurde der Code aus den Folien genommen und angepasst.
Um das *.urdf* einlesen zu können wird in der *URoboConfigFactory* auf die richtige Endung geprüft. *Formats.Add* und *FactoryCanImport* haben dazu das *urdf* Format als String.

```
URoboConfigFactory::URoboConfigFactory()
{
    // Which asset type can the factory import
    SupportedClass = URobotConfiguration::StaticClass();

    // List of supported formats. Each entry is of the form "ext;
        ↪ Description" where
    // ext is the file extension
    Formats.Add(TEXT("urdf;URDF robot description format"));

    // Factory imports objects from files
    bEditorImport = true;

    // Required the spesific other reimports do their CanReimport
        ↪ checks first
    ImportPriority = DefaultImportPriority - 1;
}

bool URoboConfigFactory::FactoryCanImport(const FString &
    ↪ Filename)
{
    return Filename.EndsWith(".urdf", ESearchCase::IgnoreCase);
}
```

FactoryCreateFile startet den Parser und generiert damit das DataAsset und setzt dort die Werte aus dem urdf File.

```
UObject * URoboConfigFactory::FactoryCreateFile(UClass * InClass,
    ↪  UObject * InParent, FName InName, EObjectFlags Flags,
    const FString & Filename, const TCHAR * Parms,
        ↪ FFeedbackContext * Warn, bool & bOutOperationCanceled)
{
```

```
    UE_LOG(LogTemp, Log, TEXT("InName %s, InParentName %s"), *
        ↪ InName.ToString(), *InParent->GetName());

    Flags |= RF_Transactional;

    // Called when new assets are being (re-)imported
    FEditorDelegates::OnAssetPreImport.Broadcast(this, InClass,
        ↪ InParent, InName, Parms);

    // Parse the .sdf buffer data into the data asset
    Parser SdfParser(Filename);

    // Create a new DataAsset
    URobotConfiguration* NewDataAsset = SdfParser.
        ↪ ParseToNewDataAsset(InParent, InName, Flags);

    // Called when new assets have been (re-)imported
    FEditorDelegates::OnAssetPostImport.Broadcast(this,
        ↪ NewDataAsset);

    return NewDataAsset;
}
```

Der Parser übernimmt die Arbeit des Übersetzen vom urdf File in Datenformate die in der Unreal Engine verwendet werden können. Dazu wird die urdf Datei in ein FXmlFile umgewandelt, welches durch Knoten durchgelaufen werden kann.

```
URobotConfiguration * Parser::ParseToNewDataAsset(UObject *
    ↪ InParent, FName InName, EObjectFlags Flags)
{
    DataAsset = NewObject<URobotConfiguration>(InParent, InName,
        ↪ Flags);

    for (const auto& ChildNode : XmlFile->GetRootNode()->
        ↪ GetChildrenNodes())
    {
        FString Tag = ChildNode->GetTag();
        FString NameAttribute = ChildNode->GetAttribute(TEXT("name
            ↪ "));

        if (Tag.Contains(FString("link")))
        {
            DataAsset->Links.Add(ParseLink(NameAttribute,
                ↪ ChildNode));
        }

        if (Tag.Contains(FString("joint")))
        {
            DataAsset->Joints.Add(ParseJoint(NameAttribute,
```

```
                ↪ ChildNode));
        }
    }

    return DataAsset;
}
```

Die Funktionen ParseLink und ParseJoint übersetzten die im urdf vorkommenden Links und Joints, so das sie dem DataAsset übergeben werden können. Diese gehen die Knoten durch und prüfen nach den Schlagwörtern aus der urdf Datei. Den Schlagwörtern entsprechend werden die Daten umgewandelt.

```
FJoints Parser::ParseJoint(const FString & Name, const FXmlNode *
    ↪  JointNode)
{
    FJoints RobotJoints;
    RobotJoints.Name = Name;
    UE_LOG(LogTemp, Warning, TEXT("Joint Name: /%s"), *Name);
    RobotJoints.Type = JointNode->GetAttribute("type");

    for (const auto& ChildAttributes : JointNode->GetChildrenNodes
        ↪ ())
    {
        //UE_LOG(LogTemp, Warning, TEXT("Joint Child Tag: %s \n"),
            ↪  *ChildAttributes->GetTag());
        //UE_LOG(LogTemp, Warning, TEXT("Joint Child Content: %s \
            ↪ n"), *ChildAttributes->GetContent());

        FString ChildTag = ChildAttributes->GetTag();
        if (ChildTag.Contains("pose"))
        {
            FString Content = ChildAttributes->GetContent();
            TArray<FString> Parts;
            Content.ParseIntoArrayWS(Parts);

            FVector Origin;
            Origin.X = FCString::Atof(*Parts[0]);
            Origin.Y = FCString::Atof(*Parts[1]);
            Origin.Z = FCString::Atof(*Parts[2]);

            RobotJoints.Pose.SetLocation(Origin);
        }
```

Nachdem das DataAsset gefüllt ist kann mit den Daten der Roboter erstellt werden. Dieser durchläuft die Arrays von Links und Joints, erstellt die jeweiligen Objekte (Formen) und setzt die Positionen.

```
void ARobot::LoadLinks(const FLink& RobotLink)
{
    FString Name = RobotLink.Name;
```

```
    UStaticMeshComponent* Link = NewObject<UStaticMeshComponent>(
        ↪ this);
    Link->SetupAttachment(RootComponent);

    if (RobotLink.Form.Contains("box"))
    {
        /**
        BoxComponent = NewObject<UBoxComponent>();;
        BoxComponent->SetWorldTransform(RobotLink.Pose);
        BoxComponent->SetWorldScale3D(RobotLink.Size);
        BoxComponent->AttachTo(Link);
        */

        Link->SetStaticMesh(Box);
        Link->SetRelativeTransform(RobotLink.Pose);
        Link->SetWorldScale3D(RobotLink.Size);
    }
```

Der Roboter kann so in die Welt geladen werden aber scheinbar ist dort bei den Positionen irgendwo ein Fehler, da alle Teile in der Body-Box sind. Der Transform ist aber entsprechend dem urdf File gesetzt und ich weiß nicht ob die Einheiten noch umgewandelt werden müssen.

Task 2

Für diese Aufgabe wurde ein Actor erstellt, welcher aus drei Boxen besteht, welche im Editor in Position gebracht wurden. Dieser Actor fährt zum *Moveable Object*, welches durch iterieren in der Welt gefunden wird. Die *Gripper* Box des Actors hat eine Overlapt-Funktion, um beim Überlappen das anderen Objekt zu binden.

```
void ACraneActor::BeginOverlap(UPrimitiveComponent *
    ↪ OverlappedComponent, AActor * OtherActor,
    ↪ UPrimitiveComponent * OtherComp, int32 OtherBodyIndex,
    ↪ bool bFromSweep, const FHitResult & SweepResult)
{
    if (!bAttached)
    {
        // Check if the right object is overlapping
        if (OtherActor->GetName().Contains("MoveableObject"))
        {
            AttachedObject = OtherActor;
            AttachedObject->AttachToComponent(Gripper,
                ↪ FAttachmentTransformRules::KeepWorldTransform);
            bAttached = true;
        }
    }
}
```

Der Actor Bewegt sich zur vorgegebenen Stelle, welche die Position des *Moveable Object* ist. Dies geschieht über die *MoveToPickupPosition* des Actors, die in jedem Tick aufgerufen wird. Diese bewegt den Actor erst in X und dann Y Richtigun zum Object hin.

```
void ACraneActor::MoveToPickupPosition()
{
    if (ObjectPosition.X < this->GetActorLocation().X)
    {
        AddActorWorldOffset(FVector(-1, 0, 0));
    }
    else if (ObjectPosition.X > this->GetActorLocation().X)
    {
        AddActorWorldOffset(FVector(1, 0, 0));
    }
    else if (ObjectPosition.Y < this->GetActorLocation().Y)
    {
        AddActorWorldOffset(FVector(0, -1, 0));
    }
    else if (ObjectPosition.Y > this->GetActorLocation().Y)
    {
        AddActorWorldOffset(FVector(0, 1, 0));
    }
}
```

Ist das Objekt *attached* wird es zur neuen Position gebracht und *detached*. Im Anschluss bewegt der Roboter sich zur Ausgangsposition zurück. Die Bewegung läuft ähnlich wie bei der *MoveToPickupPosition* Funktion ab, es kommt nur noch zusätzlich eine Rotation hinzu, die auch wieder rückgängig gemacht wird.

Note

Diese Aufgabe wurde recht statisch umgesetzt, da man die Bewegung des Actors selbst wählen konnte. Die Aufgabe wird also nicht optimal funktionieren, wenn das zu bewegende Objekt oder der Roboter ungünstig platziert werden.