2016

Light Seeking Robot

„Phototropi“

ATR-PROJEKT

AUTOR: MARK TAMAEV 516214

HS KOBLENZ | Konrad-Zuse-Straße 1,56075 Koblenz

Inhaltsverzeichnis

Bildverzeichnis

1. Einleitung

Dieses ATR Projekt ist eine Erweiterung von einem bestehenden Projekt mit einem kleinen Solarroboter. Dieser Roboter sucht den hellsten Punkt in einem Raum und richtet sich auf diesen Punkt wie eine Blume aus. Der Roboter soll so klein und robust sein, dass man ihn auf Messen und Vorträgen in der ganzen Welt mitnehmen kann. Der vorhandene Roboter ist aber noch nicht robust.

1.1 Solarroboter

Der Solarroboter ist ein selbstgebauter Hobbyroboter der nur 2 Rotationsachsen hat. Jede Achse kann sich nur um 90° drehen. Am Ende des Armes befindet sich eine Fläche mit 5 Lichtsensoren, diese Sensoren messen die Lichtstärke und geben somit die Drehrichtung an.

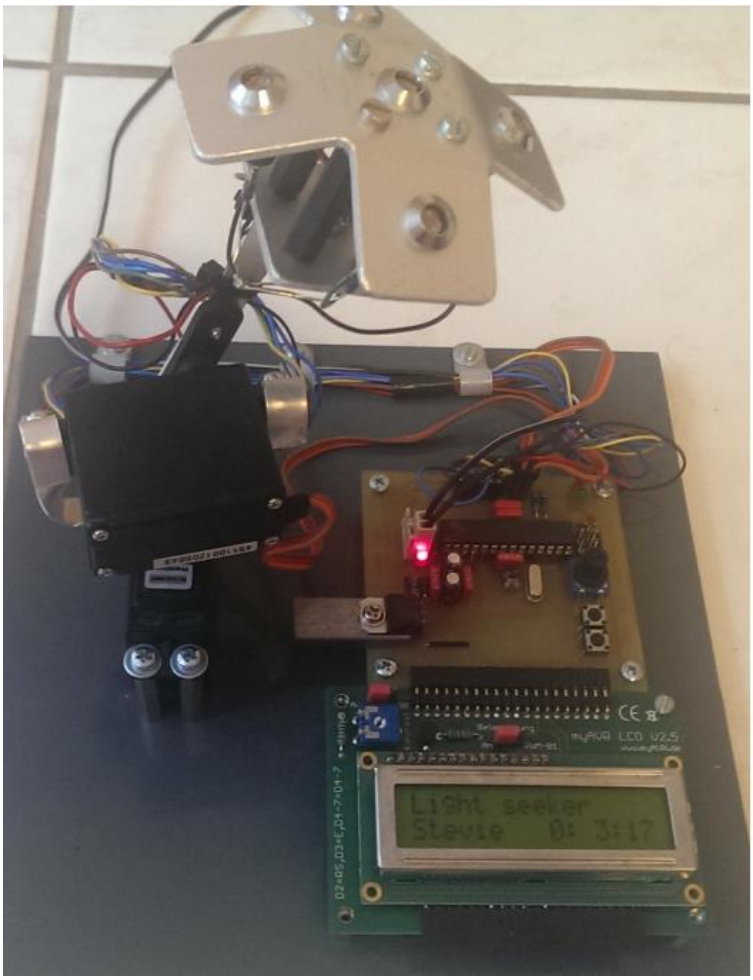

Abbildung 1 Aufbau des Light seeking Robot

Auf der Bodenplatte des Roboters befindet sich eine Platine mit einem Atmega8-Mikrocontroller, mit dem der Roboter gesteuert wird. Außerdem hat der Roboter einen 2 Zeilenbildschirm, auf dem Informationen angezeigt werden können. Zur Steuerung gibt es zwei Tasten.

1.2 Verteilung von den Aufgaben

Die Erweiterung des Projektes wurde in 4 Aufgabenbereiche aufgeteilt. Der erste Bereich befasst sich mit der Robustheit des Roboters. Für diesen Teil sollen neue, bessere Motoren angeschafft werden und ein neues Design soll entwickelt werden. Da man heutzutage die Bauteile mit einem 3D Drucker einfach drucken kann, wird alles was möglich ist mit einem 3D Drucker hergestellt.

Der zweite Bereich befasst sich mit der Portierung des alten Quellcodes auf den neuen Mikrocontroller. Der Grund dafür ist, dass der Speicher des alten Mikrocontroller voll ist und kann nicht mehr erweitert werden kann.

Der dritte behandelt die Kommunikation über eine Schnittstelle des Mikrocontrollers mit dem Computer. Die Schnittstelle übergibt die Information des Roboters an den Computer und ermöglicht somit die Steuerung.

Im letzten Teilbereich wird die Erstellung eines 3D-Modells des Solarroboters behandelt. Dieses Model empfängt die Information von dem Roboter durch die Schnittstelle und zeigt die Stellung in 3D.

1.3 Gliederung

Das folgende zweite Kapitel gibt eine Einführung in das Programm Unity 3D, dass für das 3D Modell des Roboters benutzt wurde. Dieses Programm enthält alle nötige Elemente wie Lichtquellen, geometrische Formen, Perspektiven auf die Szene und ist daher elementarer Bestandteil der hier entwickelten Anwendung.

Das dritte Kapitel beschreibt den Quellcode der Simulation. Hier wird beschrieben wie man auf die einzelnen Objekte zugreifen kann, um diese dann zu manipulieren. Auch findet man in diesem Kapitel die Beschreibung für die Kamerabewegung, die Objektsteuerung und die Schnittstelle zu dem echtem Roboter.

In dem viertem Kapitel steht die Beschreibung von dem fertigem Programm. Dieses Kapitel beschreibt die einzelnen Funktionen und Einstellungen von der Simulation.

2. Unity 3D

Unity ist eine Laufzeit- und Entwicklungsumgebung für Spiele (Spiel-Engine) die von Unity Technologies entwickelt wurde. Diese Entwicklungsumgebung wird für die Entwicklung von Computerspielen und anderer interaktiver 3D-Grafik-Anwendungen benutzt. Die Umgebung läuft auf den Betriebssystemen Windows und OS X und ermöglicht die Kompilation für die Zielplattformen wie Windows, Mac, Linux, PS3, Xbox 360, Wii, iOS, Android, PSV und Webbrowser über WebGL oder das Plug-in: Unity Web Player

Wegen der einfacher Oberfläche und der Vielzahl an Möglichkeiten wird es für 3D Simulationen und weitere wissenschaftliche Zwecke benutzt. Außerdem funktioniert es mit der Visual Studio Umgebung und arbeitet mit der Monoversion von C#. Leider arbeitet es nur mit .NET 2.0 und .NET 2.0 Subset. Dies erschwert die Programmierung von manchen speziellen Funktionen. Außer C# ermöglicht es auch die Programmierung mit JavaScript, und BOO. Wenn man kein Visual Studio benutzen kann, kann man auch MonoDevelop als Entwicklungsumgebung benutzen.

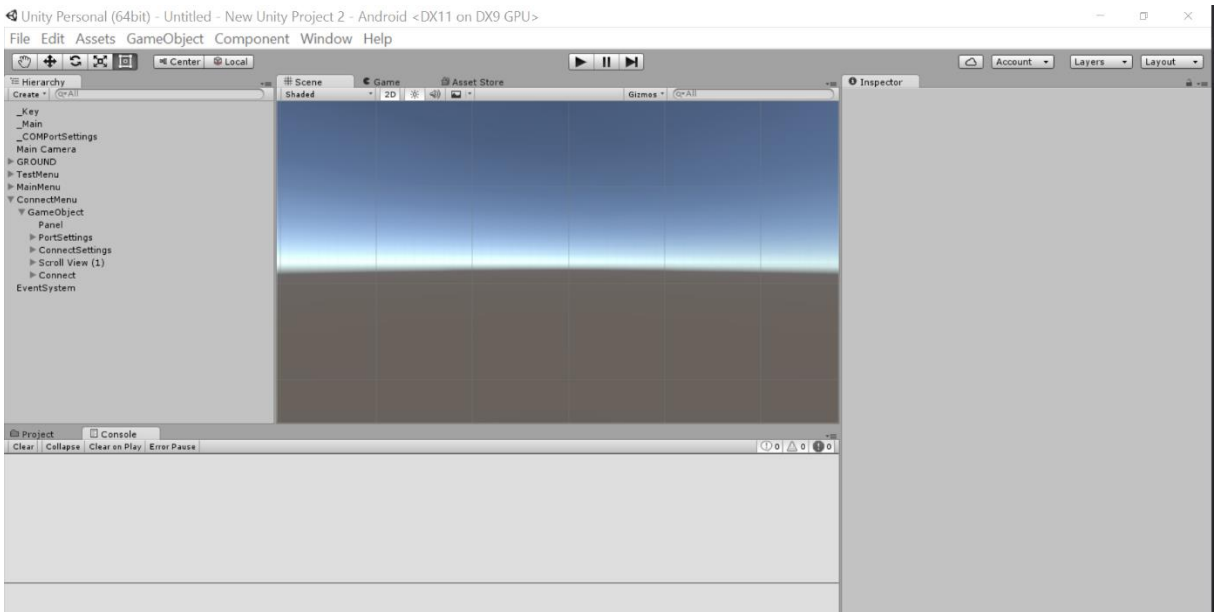

Abbildung 2 Unity Interface

Das Layout beim Unity ist frei gestaltbar. Man kann jedes Unity-Fenster aus dem Programm herauslösen und sich dahin bewegen, wo es dem User am besten passt. Das ermöglicht die vereinfachte Arbeit mit der Umgebung und es ergibt sich die Möglichkeit auf mehreren Bildschirme zu arbeiten.

2.1 Hierarchie Fenster

In diesem Bereich werden die in der Scene benutzten Objekte gezeigt. Jedes Objekt kann eine Parent – Child Beziehung haben. Die Unterobjekte, die in einem Oberobjekt liegen sind Child-Objekte. Durch diese Beziehung wird das Koordinatensystem von dem Child-Objekt an das des Parent angepasst. Sonst muss man die Koordinaten durch eine Formel umrechnen.

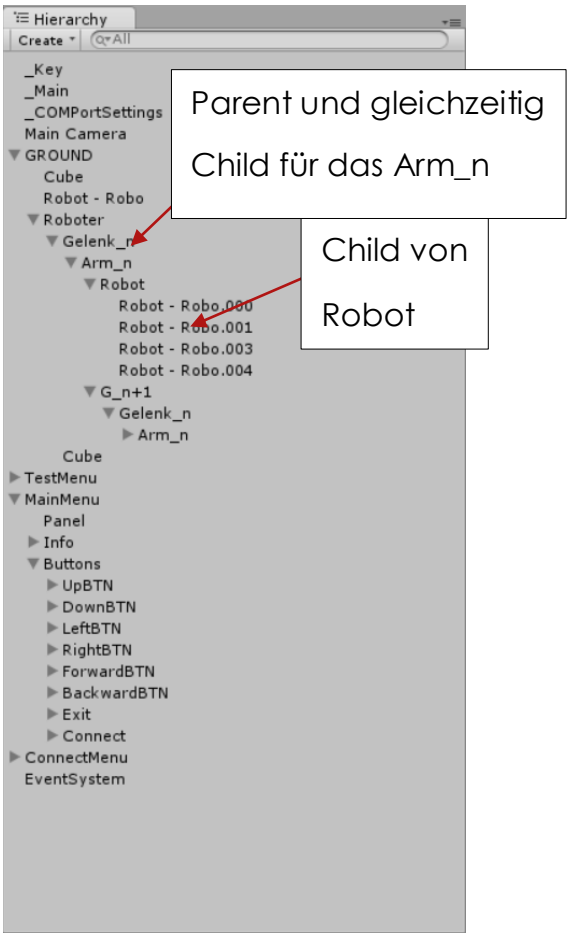

Abbildung 3 Hierarchie Fenster

In der Hierarchie werden 8 Objektarten angelegt.

1. Empty: ist ein Leeres, unsichtbares Objekt, dass man sich als durchsichtbare Tüte vorstellen kann. Dieses Objekt kann man als Platzhalter benutzen, oder wenn mehrere Objekte miteinander verbunden sind (dann funktionier es als ein Gruppe von Objekten). So kann man die Bewegungen mit Hilfe von einfachen Objekten vorprogrammieren und später, wenn das Design fertig kann man die einfachen Objekte durch komplizierte Objekte ersetzen.

2. 3D Object: Ist eine Gruppe von einfachen geometrischen Objekten wie Würfel, Kegel oder 3D Text. Diese Objekte kann man sehen und sie haben bestimmte Eigenschaften, wie z.B. eine Kollisionserkennung.

3. 2D Object: Ist ein Sprite Objekt. In dieses Objekt kann man Bilder und Videos laden. Dieses Objekt wird immer nur in 2D dargestellt und wird nie in die Tiefe gedreht (nur X und Y Achse).

4. Light: Dieses Objekt enthält alle Beleuchtungsarten. Die 3D Figuren brauchen immer Licht, sonst kann man diese nicht sehen. Die wichtigsten Lichtarten sind Spot und Directional Light. Spotlight scheint wie ein Projektor (in der Mitte gibt es Licht, außen aber nicht). Directional Light ist wie die Sonne, es leuchtet in eine Richtung.

5. Audio: Dieses Objekt enthält Audiosignale, es ist mehr für Spiele interessant. Für das Projekt ist es aber sinnlos und kann nur als Feinschliff eingefügt werden z.B. der Roboter soll bei der Drehung mechanische Geräusche rausgeben wie im Film Robocop (1987).

6. UI: Das User Interface ist ein wichtiges Objekt für uns. Es erstellt die Knöpfe, und gibt den Text raus. Es funktioniert ähnlich wie 2D Objekte.

7. Particle System: Simuliert das Verhalten von kleinen Teilchen wie Feuer, Nebel, Rauch, Haare, Staub usw.. Es ist wie die Audioobjekte für das Projekt eher sinnlos und kann als Feinschliff z.B. für die Fehlerausgabe (Rauch, wie wenn ein elektrisches Element durchbrennt.) benutzt werden. Das Particle System kann zur besseren Dekoration Bestandteil sein (Sonnenstrahlen, Wolken am Himmel, Regen).

8. Camera: Die Kamera ist wie das Licht, ein wichtiges Objekt in der 3D Szene. So können wir wie durch eine echte Kamera sehen was in der Szene passiert. Man kann den Blickwinkel ändern oder auch eine feste Position festlegen.

2.2 Inspektor Fenster

Das inspektor Fenster zeigt die Einstellungen der Objekte in dem Programm. Die meist gezeigte Einstellungen gehören zu den Objekten im Hierarchie-Fenster. Es zeigt aber auch das Projekt, den Player, die Input Settings und Andere.

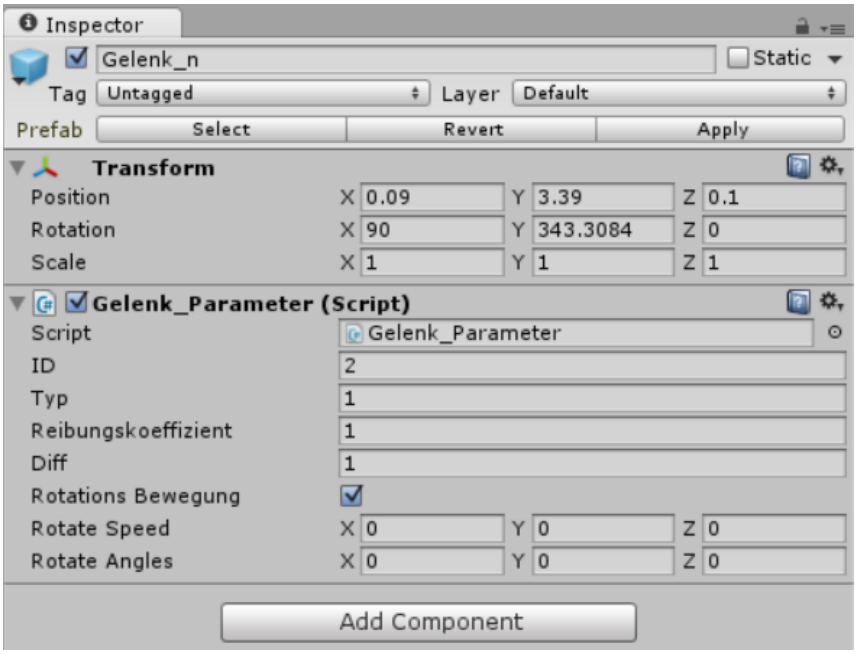

Abbildung 4 Inspektor Fenster

In den Einstellungen für das Objekt steht fast immer ein Kopf. Hier kann man den Namen für das Objekt ändern (im Bild - „Gelenk_n“) sowie auch das Icon festlegen (der Quader links von dem Namen). Mit dem Häkchen, links von dem Namen, kann man das Objekt verstecken. So kann man unbrauchbare Objekte verstecken und somit einfacher an den Details der Szene arbeiten. Auch hier kann man den Tag und den Layer festlegen. Mit diesen Einstellungen ist es einfacher die Objekte zu suchen und voneinander zu trennen. Das Static Häkchen ermöglicht den statischen Zugriff auf das Objekt.

Hier werden die wichtigen Objekte beschrieben und Erklärt:

1. Transform: Die Einstellungen, die das Bewegen eines Objektes ermöglicht. Es existiert im jedem Objekt, dass in der Szene benutzt wird. Position zeigt die Stellung des Objektes im 3-Dimensionalem Raum. Rotation dreht das Objekt und Scale vergrößert oder verkleinert es.

2. Mesh Filter: Es beschreibt die Mesh-Struktur. Es existiert in jedem geometrischem Objekt, sowohl auch komplizierten Objekten (z.B. Kopf, Baum). Man kann es für die Kollisionserkennung benutzen.

3. Mesh Renderer: Zeigt die Einstellungen die das Aussehen eines Objektes ändern. Mit diesen Einstellungen kann man festlegen ob Schatten oder Reflektion gezeigt werden. Hier kann man dem Objekt ein Material zuordnen.

4. Script: Dies wird verwendet um ein Script anzubinden und die globalen statischen Variablen festzulegen. Dieses Feld unterscheidet sich je nach Script.

5. **Texturen Inspektor:** Hier stehen die Einstellung für die einzelne Texturen.

Hier sind noch ein Paar wichtige UI Elemente, die man im Menu Bereich finden kann:

6. **Canvas:** In diesem Bereich kann man die Position von dem Menu auswählen. Sort Order gibt die Position von dem Objekt in der Tiefe (was wird von was bedeckt) an. Render Mode ist die Anknüpfung für:
    a. den Bildschirm – es wird immer auf dem Bildschirm gezeigt
    b. die Kamera – es wird immer an eine bestimmte Kamera gezeigt
    c. die Welt – es steht irgendwo in der Welt (schwebend in der Luft mit einem 3D Effekt).

7. **Canvas Scaler**: Die Fläche passt zur Bildschirmauflösung. Man stellt hier die Ausgangsauflösung ein, durch dieser Einstellung kann man das Programm später ohne Probleme und Verzahnungen skalieren.

2.3 Projekt Fenster

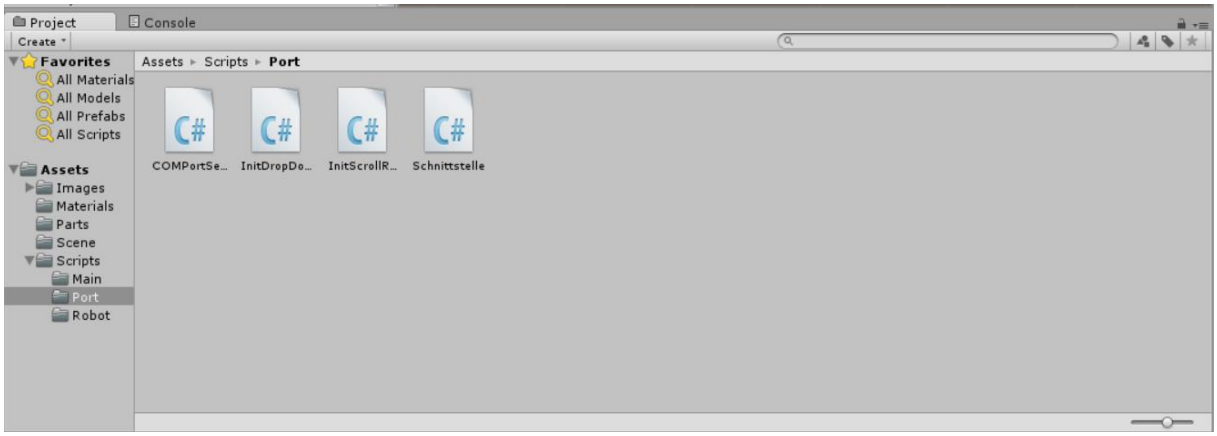

Abbildung Projekt Fenster

Dieses Fenster ist ein Fileexplorer, dass die Daten, die im Projekt benutzt werden, aufgelistet. Hier kann man die Daten, wie Bilder, Skripte, Animationen, Materiale, Audio/Videodaten und viele andere Arten von Dateien finden.

1. **Script:** Sind mit JavaScript oder C# geschriebene Skripte, die das Verhalten von Objekten und andere Funktionen beschreiben. In jedem Standartskript werden immer 2 Funktionen aufgelistet. Start ist die Initialisierung von dem Objekt und Update ist eine Funktion, die mehrmals aufgerufen wird.

2. **Bilder-, Video- und Audiodaten:** Sind die im Projekt benutzten Bilder-, Video- und Audiodaten. Man kann diese als Hintergrund abspielen/zeigen lassen oder auch als Texturen benutzen.

3. **Materiale:** Beschreiben die Oberfläche von dem Objekt. Somit kann man glänzende oder transparente Materiale erstellen. Hier kann man auch das Verhalten der Materie beschreiben wie z.B. die Bewegung von Wasseroberflächen, insbesondere wie diese sich durch Berührung verändern so dass z.B. Kreiswellen entstehen.

4. **Animationen:** Sind Bewegungen des Objektes. Mit dieser Datei kann man schwierigere Bewegungen erstellen wie z.B. die Bewegungen von Händen und Füßen oder vorprogrammiertes Verhalten der Objekte.

5. **Folder:** Die Ordner erleichtern die Übersicht von Daten und sie funktionieren genauso wie die Ordner im Explorer im Windows, Finder im OSX oder dem Nautilus im Linux.

2.4 Konsolenausgabe Fenster

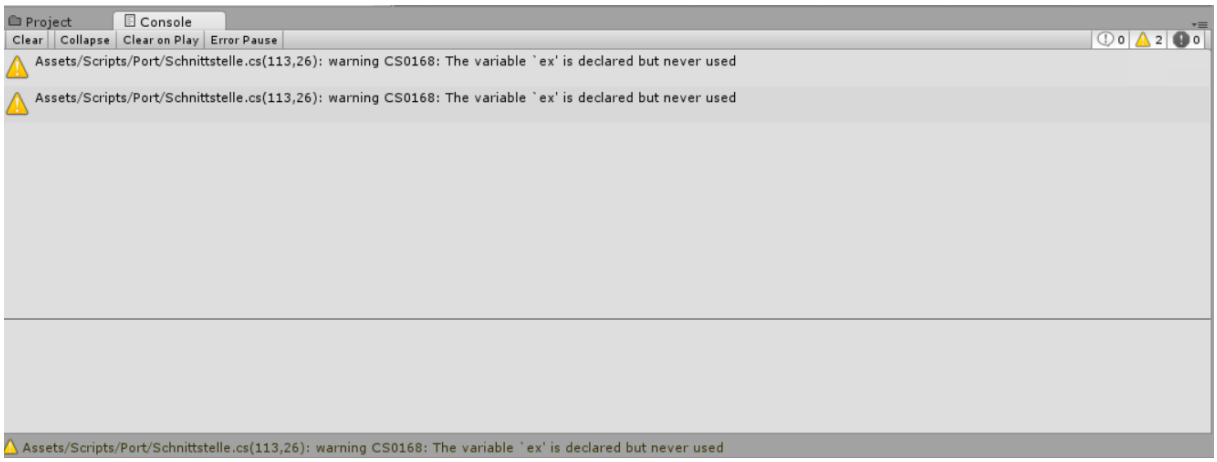

Abbildung 5 Konsolenausgabe Fenster

Im dem Konsolenfenster werden die Fehler und Warnungen angezeigt. Es gibt auch die Konsolenausgabe wie println aus. Es funktioniert genauso wie die Konsolenausgabe im Visual Studio oder Eclipse.

2.5 Scene und Game Fenster

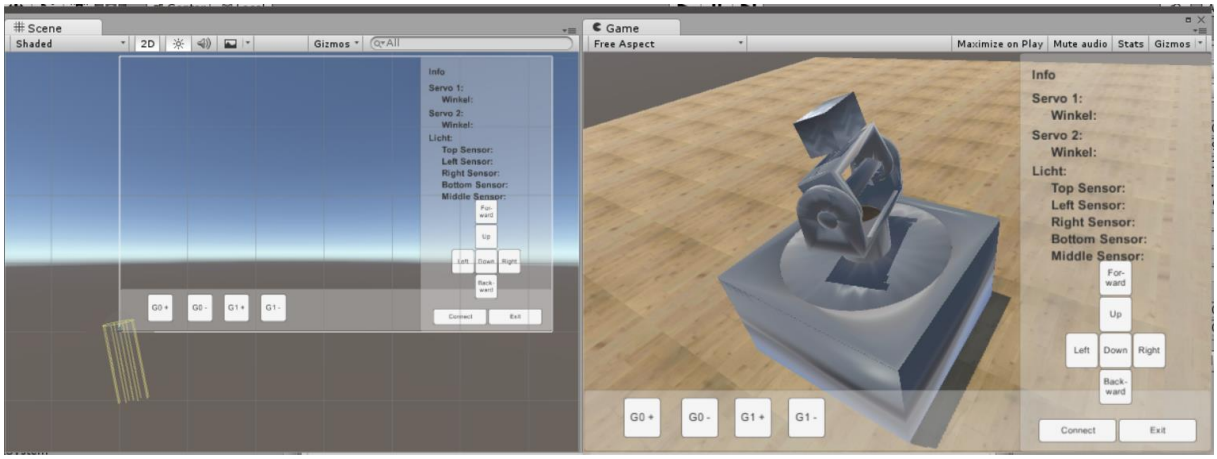

Abbildung 6 Scene und Game Fenster

In dem Szenenfenster wird die 3D Szene hergestellt. Hier kann man die Objekte selektieren und bewegen. Da wo das Wort Shaded steht wird die Anzeigeneinstellung geändert. Man kann somit entweder das Liniendesign oder das 3D Design mit Schatten auswählen. Diese Funktion erleichtert die Bearbeitung von der Szene, da man durch die Linien besser dem versackten Objekte sehen kann als mit Shaded.

2D verändert die Szene von 3D ins 2D Mode. Es erleichtert die Bearbeitung von 2D Objekten, wie z.B. das Menu.

Mit **Rechter Maustaste** kann man die Szene drehen.

Mit **Linker Maustaste** bewegt und selektiert man die Objekte.

Mit der **Scroll taste** kann man das Bild vergrößern und klicken und bewegen.

Mit den **Pfeiltasten** kann man durch die Szene gehen.

Mit dem Koordinatensystem oben rechts kann man die Szene in die Top, Front und weitere Hauptrichtungen drehen. Der Text unter dem Koordinatensystem ändert die Szene in 2 verschiedene Sichten. Die eine ist isometrisch und die andere ist perspektivisch. Hierbei ist zu beachten, dass bei der isometrischen Sichtweise die Genauigkeit höher ist.

2.6 Build Einstellungen

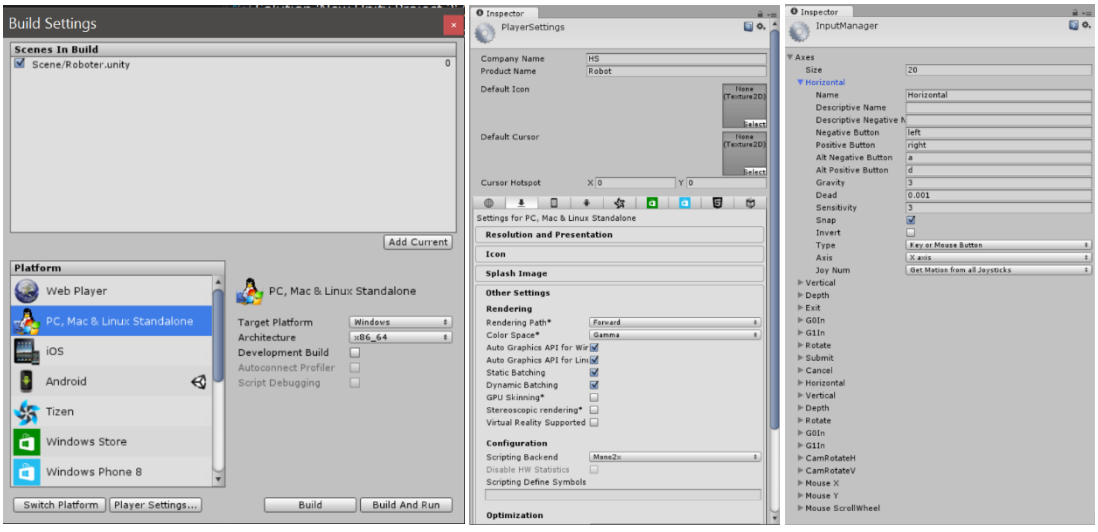

Abbildung 7 Build Einstellungen

Build Einstellungen (Abbildung 8 Links): Hier kann man das Laufsystem auf dem man später die Simulation starten will auswählen. Wenn man es für Android kompiliert hast, dann muss man hier Android als Plattform auswählen und für diese Plattform die Parameter setzen.

Player Settings (Abbildung 8 Mitte): Hier stehen die App-Einstellungen für bestimmte Plattformen. Man muss aber beachten, dass die Kompatibilität zu dieser Plattform passt z.B. muss das Default API Level auf .NET 2.0 Subset eingestellt werden. Um die .NET 2.0 Entwicklungsumgebung voll auszunutzen zu können, muss hier .NET 2.0 anstatt .NET 2.0 Subset ausgewählt werden. Weitere interessante Einstellungen die man hier finden kann, sind unter anderem die Einstellungen für den Programnamen, Icon und Startbild. Die restlichen Einstellungen sind plattformabhängig.

Input Manager (Abbildung 8 Rechts): Dieses Menu liegt in `Edit->Project Settings->Input`. Dort kann man Key-Kombinationen zuweisen. Diese werden dann beim Starten von dem Programm gezeigt. Man kann aber auch die Keys ohne dem Input-Manager an die Funktionen anbinden. Dies hat aber ein paar Nachteile, wie z.B. die Tastenänderung und in dem Manager kann man den Tasten ein Name zuweisen, dies erleichtert die Programmierung.

3 Codebeschreibung

In diesem Abschnitt werden für das Projekt wichtige Skripte und Funktionen beschrieben.

3.1 Objekt zugriff

Im Unity kann man auf die Objekte in der Szene mit Hilfe der Funktionen `GetComponent, GetComponentInChildren und GetComponentInParent` zugreifen. Dadurch können aber manchmal sehr lange und unübersichtliche Aufrufe entstehen wie z.B. `GetComponentInChildren(„GameObjekt“).GetComponentInChildren(„GameObjekt“)…` usw.. Um solch lange Kodezeilen zu vermeiden, habe ich in der Klasse Main die static Variablen angelegt, die die wichtigen Objekte enthalten. Für jedes Objekt, das durch ein Script verändert werden soll, wurde eine Funktion geschrieben. Dadurch verweist die Variable direkt auf das Objekt. Durch diesen Umweg kann man einfacher unter Zuhilfenahme der Main Klasse () auf die Objekte zugreifen.

Beispiel (aus der KeyInput->Update Funktion)**:**

```
Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y = Input.GetAxis("Rotate");
```

3.2 Objekt Bewegung

Da Unity die Koordinatensysteme für die Childobjekte (die Objekte die zu einem „Parentobjekt“ gehören) selbst berechnet, braucht man hier keine große Rechnungen zu betreiben. Man muss nur die Gelenke des Roboters bewegen oder drehen, dadurch werden die angebundenen Objekte dann mit bewegt.

Im folgenden Codeabschnitt aus der Klasse Gelenk-Parameter wird die Objektbewegung gezeigt. Wenn der Roboter nicht verbunden ist, kann sich der simulierte Roboter durch die Tasteneingaben in alle Richtungen bewegen. Hierbei sorgt der Variablen-Vektor `RotateSpeed` für die Drehrichtung. Dann wird die Position des Roboters übernommen in die er sich gerade befindet.

```
        if (!COMport.isConnected())
        {
            transform.Rotate(RotateSpeed);
        }
        else
            transform.eulerAngles = RotateAngles;
```

Die Funktionen `G0MDown`, `G1MDown`, `G0PDown` und `G1PDown` ändern den `RotateSpeed` Vektor und geben somit die Rotationsbewegung an.

3.3 Free Fly Camera (Frei fliegende Kamera)

Leider hat Unity kein eigenes Skript, dass die Kamerabewegung ermöglicht. Dafür kann man es einfach selbst programmieren. Dafür muss man mehrere Fälle betrachten.

1. „WASD"-Bewegung

In diesem Fall wird die Kamera wie in einer Spielfigur über die X und Z Achse bewegt (.). Da in Spielen für solche Bewegungen öfter als Standard die W, A, S, D Tasten benutzt werden heißt es auch hier „WASD"-Bewegung. In dieser Tastenbelegung werden die Tasten W und S für die Z-Achse benutzt und A und D für die X-Achse (man kann auch die F und R Tasten für die Y-Achse benutzen)

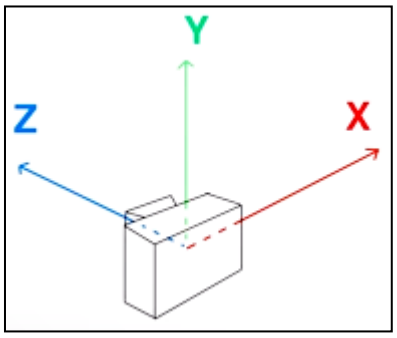

Abbildung 8 Kamera Koordinaten

Im **Kode**:

Mit der Vector3 Funktion wird ein Vektor angelegt, der die Richtung in die Kamera geht angibt.

```
        Vector3 dir = new Vector3();

         dir.x = KeyInput.Horizontal();
         dir.y = KeyInput.Depth();
         dir.z = KeyInput.Vertical();
```

Die Funktionen `Horizontal`, `Depth` und `Vertical` geben eine Zahl an, die zwischen -1 und 1 liegt wenn eine „WASD" Taste gedrückt wurde und 0 wenn keine Taste gedrückt wurde.

Die Funktion `Normalize` setzt den Betrag des Vektors auf 1. So kann man die Geschwindigkeit von der Kamera im Programm anpassen. Die Funktion `transform.Translate` bewegt das Objekt in die Richtung des Vektors. Mit `speed und Time.deltaTime` wird die Geschwindigkeit (Betrag des Vektors) der Kamera angepasst.

```
        dir.Normalize();

        transform.Translate(dir * speed * Time.deltaTime);
```

2. Mouse Look

Außer der Bewegung brauchen wir auch die Kamera Rotation. Dafür wird der Kamerawinkel an den Mauszeiger gebunden. Um die Drehrichtung zu bestimmen müssen wir nur den Vektor der zwischen der alten und der neuen Zeigerposition anliegt herausfinden.

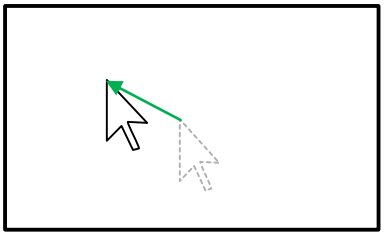

Abbildung 9 Mausvektor

Im **Code**:

Mit Hilfe der Funktionen `RotateCamerainXPos und RotateCamerainXPos` aus der Klasse `KeyInput` kann man die X und Y Richtungen des Mauszeigervektors bekommen. Die Funktion `transform.localEulerAngles` dreht die Kamera in die Richtung des Vektors. Die Winkel werden in Grad eingegeben.

```
        float newRotationX = KeyInput.RotateCamerainXPos(transform.localEulerAngles.x);
        float newRotationY = KeyInput.RotateCamerainYPos(transform.localEulerAngles.y);

        transform.localEulerAngles = new Vector3(newRotationX, newRotationY, 0);
```

3.4 Schnittstellen

Um mit dem Roboter zu kommunizieren wurden zwei Skripte geschrieben. Das erste heißt COMPortSettings. Es verbindet die GUI der Simulation mit dem COM Port. Die eigentliche Kommunikation ist in der Klasse COMport beschrieben. Die Trennung ermöglicht eine unabhängige Veränderung von GUI und dem seriellem Port, sowohl auch die Möglichkeit die Kommunikation zwischen dem Roboter und Simulation später durch andere Kommunikations-algorithmen zu ersetzen.

Die Funktion `COMport.getPortNames()` sucht nach existierenden Ports im Computer und liefert die gefundenen Ports als ein String Array. Diese Funktion muss bei der Initialisierung abgerufen werden.

Die Funktion `COMport.Connect(String port)` stellt die Verbindung mit dem Roboter her und startet die Kommunikation. In dem Parameter `port` soll der Name von dem Port eingegeben werden mit dem die Verbindung hergestellt wird.

Um die Verbindung zu unterbrechen wird die Funktion `COMport.Disconnect()` benutzt.

Die Funktion `COMport.ReceiveData()` empfängt die Daten von dem Roboter. Außerdem sortiert sie und speichert diese Information in den dazu angelegten Variablen:

```
switch (data[0])
{
    case "L0": Light[0] = double.Parse(data[1]); break;
    case "L1": Light[1] = double.Parse(data[1]); break;
    case "L2": Light[2] = double.Parse(data[1]); break;
    case "L3": Light[3] = double.Parse(data[1]); break;
    case "L4": Light[4] = double.Parse(data[1]); break;

    case "S0": ServoAngle[0] = double.Parse(data[1]); break;
    case "S1": ServoAngle[1] = double.Parse(data[1]); break;
    default: info = input; break;
}
```

Da die Übertragung durch die Schnittstelle viel Zeit in Anspruch nimmt und somit verlangsamt die Simulation von dem Roboter, wird die Funktion `ReceiveData()` in einem Thread aufgerufen.

Die Funktionen `getServoAngle(int ID)`, `getLight(int ID)` und `getInfo()` liefern die letzten empfangenen Werte. Mit ID kann man den Servo oder den Lichtsensor auswählen. Info liefert unsortierte Information von dem Roboter.

Die Funktionen `string[] getLog()` und `string getLastLog(int numberOfLogs)` liefern den Verbindungslog. Mit Hilfe von diesen Funktionen kann man die Fehler besser nachvollziehen.

4. 3D Roboter Simulation

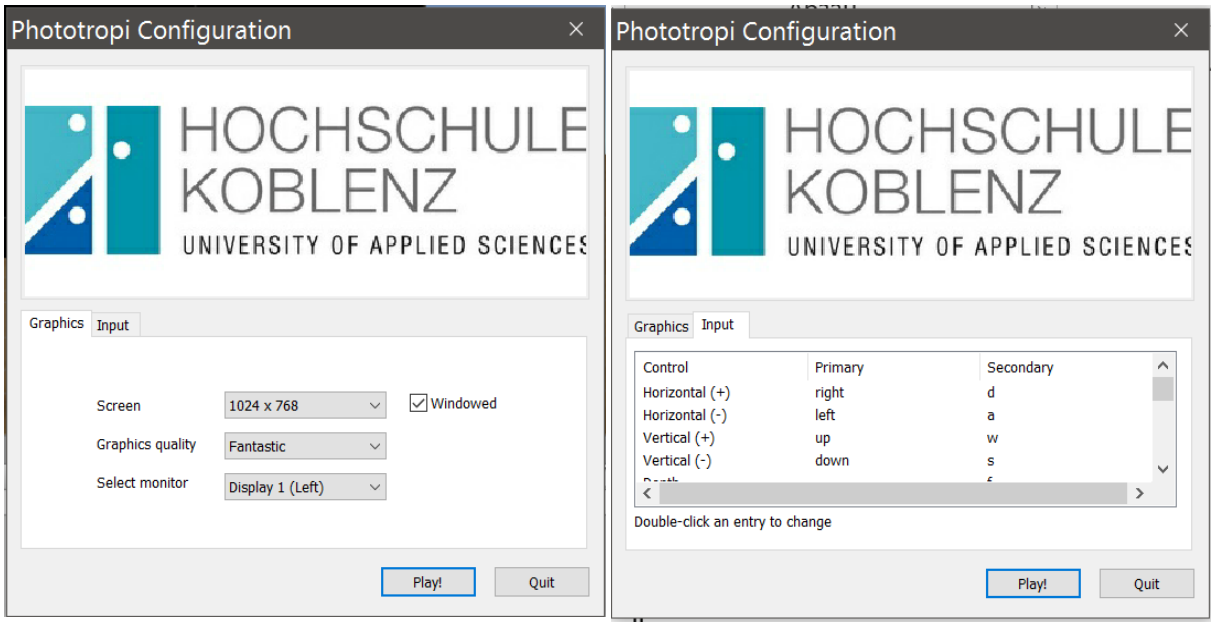

Abbildung 10 Phototropi Voreinstellungen

Beim Start des Phototropi Simulators erscheint ein Einstellungsfenster (Abbildung 10). Hier kann man auswählen ob man den Simulator in Fullscreen öffnen will oder in einem Fenster. Um die Performance bei einem schwächeren Computer zu verbessern kann man hier auch die Grafikqualität heruntersetzen.

Im Input Tab kann man die Tastenbelegung ändern. Dafür muss man die gewünschte Funktion zweimal klicken und dann die gewünschte Taste drücken. Für eine Funktion kann man zwei Tasten belegen, wie z.B. horizontale Bewegung in Plus-Richtung: Pfeiltaste „Rechts“ und Taste „D“.

Nachdem alles eingestellt wurde, kann mit dem Drücken auf die Taste „Play!“ die Simulation gestartet werden. Die Einstellungen werden gespeichert und bei dem nächsten Start wieder übernommen.

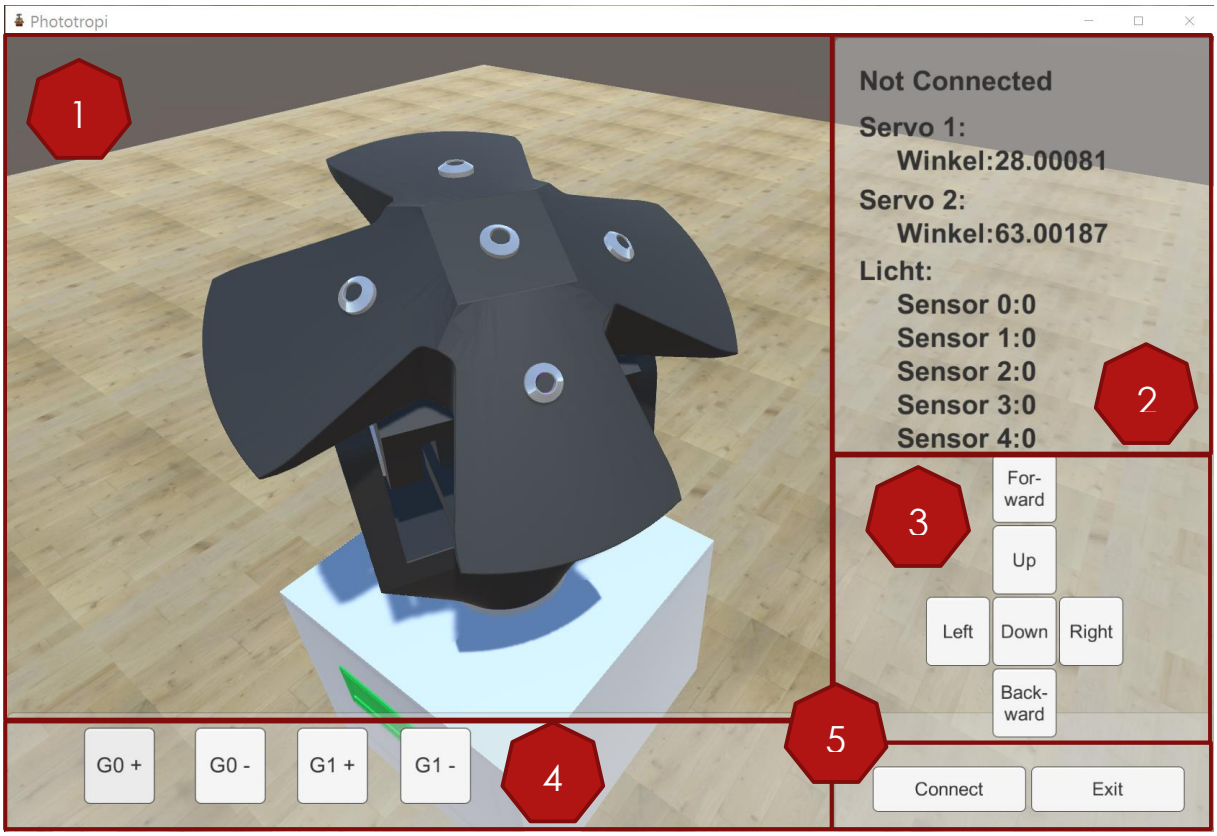

Abbildung 11 Phototropi Simulationsfenster

Das Simulationsfenster (Abbildung 11) ist das Hauptfenster in dem Programm. Es unterteilt sich in 5 Abschnitte.

In dem ersten (1) Abschnitt wird das 3D Model gezeigt. Mit Rechtsklick + Mausbewegung oder durch berühren von zwei Fingern auf dem Touchscreen kann die Kamera gedreht werden. Die Knöpfe WASDRF bewegen die Kamera (die Tasten können sich je nach Einstellung unterscheiden).

Der zweite (2) Abschnitt zeigt die Information über den Roboter. Im oberen Teil wird die Information über den Zustand des Roboters angezeigt. Servo 1 und 2 zeigt den Winkel in Grad in die sich der Roboter gedreht hat. Licht zeigt die Lichtstärke von jedem Sensor.

Im Abschnitt drei (3) werden die Knöpfe für Kamerabewegung gezeigt, für den Fall wenn die Tastatur nicht vorhanden ist, wie z.B. bei arbeiten mit einem Smartphone.

Die Knöpfe im Abschnitt 4 sind für die Testbewegungen des Roboters zuständig. Diese Knöpfe funktionieren nur wenn der Roboter nicht verbunden ist. G0 (Gelenk 0) dreht den Servo, der auf der Bodenplatte angebunden ist. G1 dreht den zweiten Servo. Für diese Knöpfe gibt es die Tastenkombination: Auf dem Numpad – ‚0'+'+'/'-' um den ersten Servo zu bewegen und ‚1'+'+'/'-' um den Zweiten zu bewegen.

In dem verbleibenden Abschnitt 5 befinden sich die Knöpfe „Connect" und „Exit". Mit dem „Exit" Knopf kann man die Anwendung schließen (es ist bequemer wenn das Programm im Fullscreen gestartet wurde). Der „Connect" Knopf öffnet das Verbindungsmenu (Abbildung 12), dass die Einstellungen für die Schnittstelle enthält.

Abbildung 12 Verbindungsmenu

Um die Simulation mit dem Roboter zu verbinden, wird im Hauptfenster der „Connect" Knopf gedrückt. Dann wird im Verbindungsmenu der Port auswählt (der Port soll kleiner als COM10 sein) mit dem eine Verbindung hergestellt werden soll. Im Dropdown Menü Connect muss „True" ausgewählt werden um die Verbindung zu starten. Wenn die Verbindung getrennt werden soll, dann muss Connection „False" gestellt werden.

Im rechten Teil werden die angehende und ausgehende Information und die Übertragungsfehler gezeigt.

Zusammenfassung

In dieser Arbeit wurde ein dreidimensionales Modell des tragbaren Solarroboters mit Hilfe der Unity 3D Umgebung und der C# Programmiersprache erstellt.

Im ersten Arbeitsschritt wurde die 3-D-Welt des Roboters hergestellt. Es wurden Licht, Kamera und Bodenplatte innerhalb des Modells auf die richtige Stelle positioniert. Für die Kamera wurde ein Skript geschrieben, dass die Kamera mit Hilfe von Tasteneingabe bewegt.

Nachfolgend entstand die Implementierung eines Benutzerinterface, dass die Werte ausgibt und die Knöpfe enthält, die die einfache Arbeit mit dem Programm ermöglichen. Gleichzeitig wurden die Input-Parameter eingestellt und die Eingaben an die Simulation angepasst.

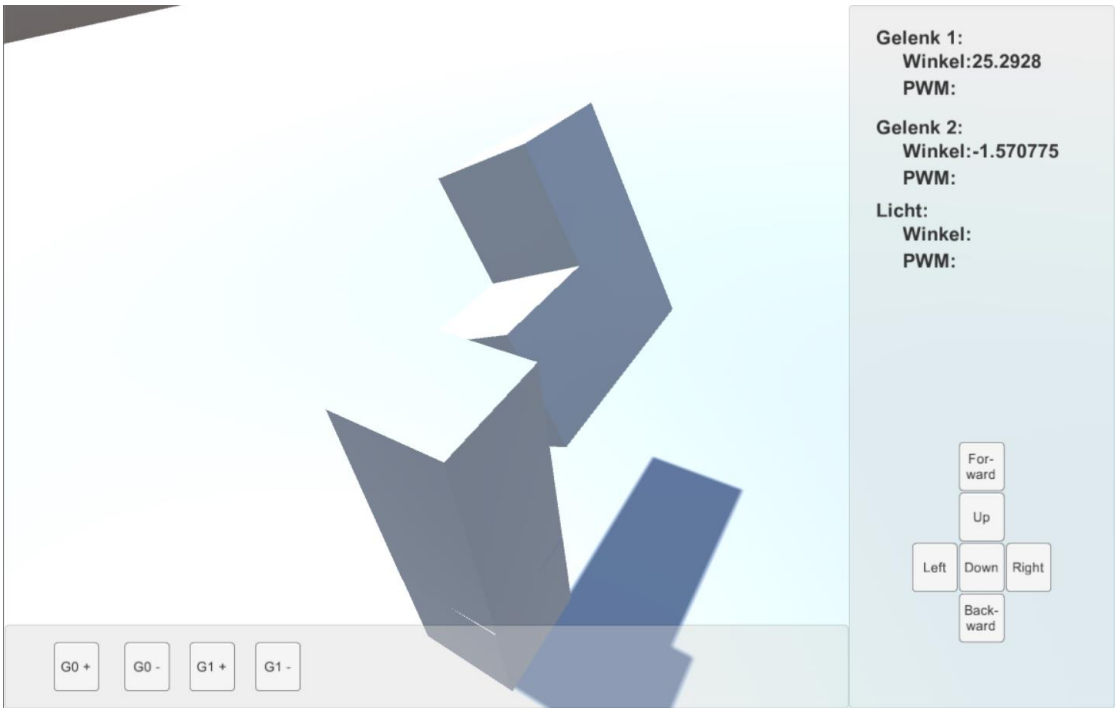

Abbildung 13 Die Erste Variante von dem Programm

Im nächsten Schritt wurde eine 3-D Struktur entworfen, um das einfache Programmieren der Robotersimulation zu ermöglichen. Dieser Schritt wurde so programmiert, dass man am Ende die Möglichkeit hat das fertige 3-D Modell schnell zu ersetzen. In diesem Schritt wurde für das Modell die einfachen Elemente Quader und Kugel benutzt.

Nachdem die 3-D Struktur fertiggestellt wurde, folgte als nächstes das Erstellen der Skripte für die Bewegung der Objekte. Dabei ist die Bewegung der Lichtquelle an den Roboter gebunden und diese erzeugt eine realistische Schattenbewegung.

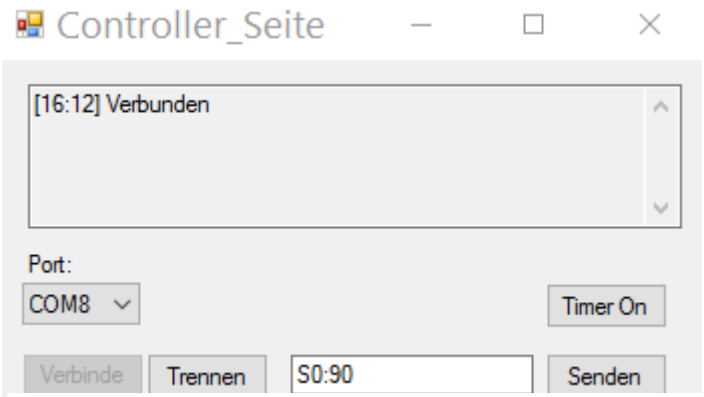

Abbildung 14 COM Port Teste Programm

Anschließend war die Simulation bereit, darauf folgte die Einbindung der COM Port Schnittstelle. Diese Schnittstelle wurde an die Schnittstelle des Roboters angepasst und mit Hilfe von Testprogrammen, (Abbildung 14) die den Übertragungsverhalten simulieren, getestet. Anschließend wurde die Schnittstelle an dem fertigen Roboter getestet und angepasst.

Abbildung 15 Roboter Design

Nach dem das Enddesign von dem Roboter bekannt wurde, wurde das 3-D Modell von dem fertigem Roboter in die Simulation eingefügt und an die vorhandene Struktur angepasst.

Literaturverzeichnis

https://www.youtube.com/watch?v=D-DjR12WsZo

https://de.wikipedia.org/wiki/Unity_(Spiel-Engine)

Anhang

Main Funktionen:

Main.cpp

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Main : MonoBehaviour
{
    public static List<Transform> Gelenk = new List<Transform>();
    public static List<Transform> Arm = new List<Transform>();
    public static GameObject ConnectionMenu;


    // Use this for initialization
    void Start(){}

    // Update is called once per frame
    void Update(){}
}
```

KeyInput.cpp

```
using UnityEngine;
using System.Collections;

public class KeyInput : MonoBehaviour
{
    static bool UpBTN = false;
    static bool DownBTN = false;
    static bool LeftBTN = false;
    static bool RightBTN = false;
    static bool ForwardBTN = false;
    static bool BackBTN = false;

    private bool isConnectionShowing=false;

    // Use this for initialization
    void Start()
    {
        Main.ConnectionMenu.SetActive(isConnectionShowing);
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetAxis("Exit") >= 0.5)
            EXIT();

        if (Input.GetAxis("G0In") >= 0.5)
            Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y            =
Input.GetAxis("Rotate");
        if (Input.GetAxis("G1In") >= 0.5)
            Main.Gelenk[1].GetComponent<Gelenk_Parameter>().RotateSpeed.y            =
Input.GetAxis("Rotate");

    }

    public void EXIT()
```

```
    {
        Application.Quit();
    }

    public void G1PDown()
    {
        Main.Gelenk[1].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 1;
    }

    public void G1MDown()
    {
        Main.Gelenk[1].GetComponent<Gelenk_Parameter>().RotateSpeed.y = -1;
    }

    public void G0PDown()
    {
        Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 1;
    }

    public void G0MDown()
    {
        Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y = -1;
    }

    public void G1PUp()
    {
        Main.Gelenk[1].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 0;
    }

    public void G1MUp()
    {
        Main.Gelenk[1].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 0;
    }

    public void G0PUp()
    {
        Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 0;
    }

    public void G0MUp()
    {
        Main.Gelenk[0].GetComponent<Gelenk_Parameter>().RotateSpeed.y = 0;
    }

    public static float RotateCamerainXPos(float x)
    {
        return x - Input.GetAxis("Mouse Y");
    }

    public static float RotateCamerainYPos(float y)
    {
        return y + Input.GetAxis("Mouse X");
    }

    public static bool isMouseUp()
    {
        return Input.GetMouseButtonUp(1);
    }

    public static bool isMousDown()
    {
        return Input.GetMouseButtonDown(1);
    }
```

```
    public static bool isDown()
    {
        if (DownBTN)
            return true;
        return Input.GetAxis("Depth") <= -0.5;
    }

    public static bool isUp()
    {
        if (UpBTN)
            return true;
        return Input.GetAxis("Depth") <= 0.5;
    }

    public static bool isLeft()
    {
        if (LeftBTN)
            return true;
        return Input.GetAxis("Horizontal") <= -0.5;
    }

    public static bool isRight()
    {
        if (RightBTN)
            return true;
        return Input.GetAxis("Horizontal") >= 0.5;
    }

    public static bool isBack()
    {
        if (BackBTN)
            return true;
        return Input.GetAxis("Vertical") <= -0.5;
    }

    public static bool isForward()
    {
        if (ForwardBTN)
            return true;
        return Input.GetAxis("Vertical") >= 0.5;
    }

    public static float Vertical()
    {
        if (ForwardBTN)
            return 1;
        if (BackBTN)
            return -1;
        return Input.GetAxis("Vertical");
    }

    public static float Horizontal()
    {
        if (RightBTN)
            return 1;
        if (LeftBTN)
            return -1;
        return Input.GetAxis("Horizontal");
    }

    public static float Depth()
    {
        if (UpBTN)
            return 1;
```

```
        if (DownBTN)
            return -1;
        return Input.GetAxis("Depth");
    }

    public static float CamRotationH(float position)
    {
        return position + Input.GetAxis("CamRotateH");
    }

    public static float CamRotationV(float position)
    {
        return position + Input.GetAxis("CamRotateV");
    }

    public void setBTN(string keyAndValue)
    {
        string key = keyAndValue.Split(':')[0];
        bool value = keyAndValue.Split(':')[1].Contains("True");
        switch (key)
        {
            case "UpBTN": KeyInput.UpBTN = value; break;
            case "DownBTN": KeyInput.DownBTN = value; break;
            case "LeftBTN": KeyInput.LeftBTN = value; break;
            case "RightBTN": KeyInput.RightBTN = value; break;
            case "ForwardBTN": KeyInput.ForwardBTN = value; break;
            case "BackBTN": KeyInput.BackBTN = value; break;
        }
    }

    public void ConnectBtn()
    {
        isConnectionShowing = !isConnectionShowing;
        Main.ConnectionMenu.SetActive(isConnectionShowing);
        Main.ConnectionMenu.transform.localPosition=new Vector3(0, 0, 0);
    }
}
```

ConnMenu.cpp

```
using UnityEngine;
using System.Collections;

public class ConnMenu : MonoBehaviour {

       // Use this for initialization
       void Start () {
        Main.ConnectionMenu = this.gameObject;
       }

       // Update is called once per frame
       void Update () {}
}
```

Camera.cpp

```
using UnityEngine;
using System.Collections;

public class Camera : MonoBehaviour
{

    public float speed = 10;
    private bool down = false;
    // Use this for initialization
    void Start(){}

    // Update is called once per frame
    void Update()
    {
        Vector3 dir = new Vector3();

        dir.x = KeyInput.Horizontal();
        dir.y = KeyInput.Depth();
        dir.z = KeyInput.Vertical();

        if (KeyInput.isMousDown())
            down = true;
        if (KeyInput.isMouseUp())
            down = false;

        if (down)
        {
            float                                newRotationX                                =
KeyInput.RotateCamerainXPos(transform.localEulerAngles.x);
            float                                newRotationY                                =
KeyInput.RotateCamerainYPos(transform.localEulerAngles.y);

            transform.localEulerAngles = new Vector3(newRotationX, newRotationY, 0);
        }
        else
        {
            //game pad rotate
            transform.localEulerAngles                          =                          new
Vector3(KeyInput.CamRotationV(transform.localEulerAngles.x),
KeyInput.CamRotationH(transform.localEulerAngles.y), 0);
        }
        dir.Normalize();

        transform.Translate(dir * speed * Time.deltaTime);
        transform.Translate(dir * speed * Time.deltaTime);

    }

}
```

COM Port Funktionen

COMPortSettings.cpp

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;
using System;
using System.IO.Ports;

public class COMPortSettings : MonoBehaviour
{

    public static List<Dropdown> Dropdowns = new List<Dropdown>();
    public static List<Text> ScrollViews = new List<Text>();
    public static List<string> PortNames = new List<string>();
    private static string port = "";
    private static bool init = true;

    // Use this for initialization
    void Start()
    {
        COMport.DataReceived += DataReceived;
    }

    private void DataReceived(string data)
    {

    }

    private static void Init()
    {
        PortNames.Clear();
        PortNames.AddRange(COMport.getPortNames());
        Dropdowns[1].options.Clear();
        port = PortNames[0];
        foreach (string item in PortNames)
        {
            Dropdowns[1].options.Add(new Dropdown.OptionData(item));
        }
        Dropdowns[1].value = 0;
    }

    // Update is called once per frame
    void Update()
    {
        if (init&& Dropdowns.Count>0)
        {
            Init();
            init = false;
        }
        COMport.ReceiveData();
        ScrollViews[0].text = COMport.getLastLog(10);

    }

    public void ConnectBTN(int ID)
    {
        switch (Dropdowns[ID].value)
        {
            case 0: COMport.Disconnect(); break;
```

```
            case 1: COMport.Connect(port); break;
            default: break;
        }
    }

    public void COMBTN(int ID)
    {
        port = PortNames[Dropdowns[ID].value];
    }
}
```

Schnittstelle.cpp

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Linq;

class COMport
{
    public static List<string> Log = new List<string>();
    public static SerialPort sport;
    public delegate void Received(string data);
    public static event Received DataReceived;

    private static double[] ServoAngle = new double[2];
    private static double[] Light = new double[5];
    private static string info = "";
    private static string ReceivedTemp = "";

    public string[] getLog()
    {
        return Log.ToArray();
    }

    public static string getLastLog(int numberOfLogs)
    {
        string temp = "";
        if(Log.Count>0)
        for (int i = 0; i < (int)((Log.Count() > numberOfLogs) ? (numberOfLogs) :
(Log.Count())); i++)
        {
            temp += Log[Log.Count() - i-1];
        }
        return temp;
    }

    private static void setLog(string info)
    {
        Log.Add(DateTime.Now.ToShortTimeString() + ": " + info + Environment.NewLine);
        if (Log.Count() > 10000)
        {
            Log.RemoveAt(0);
        }
    }

    public static string[] getPortNames()
    {
        setLog("getPorts:");
        List<string> ports = new List<string>();
        foreach (String portName in SerialPort.GetPortNames())
        {
```

```
            ports.Add(portName);
            setLog(portName);
        }
        return ports.ToArray();
    }

    public static void Connect(String port)
    {
        try
        {
            setLog("Connect to: " + port);
            serialport_connect(port, 9600, Parity.None, 8, StopBits.One);
        }
        catch (Exception ex)
        {
            setLog(ex.Message);
            throw;
        }
    }

    public static bool Disconnect()
    {
        setLog("Disconnect");
        if (sport.IsOpen)
        {
            sport.Close();
            setLog("Port is Closed");
            return true;
        }
        setLog("Port was Closed");
        return false;
    }


    public static bool isConnected()
    {
        if (sport != null)
            if (sport.IsOpen)
                return true;
        return false;
    }


    public static void ReceiveData()
    {

        try
        {
            if (sport != null)
                if (sport.IsOpen)
                {
                    var tes = (char)sport.ReadChar();

                    if (tes != '\n')
                        ReceivedTemp += tes.ToString();
                    else
                    {
                        sport_DataReceived(ReceivedTemp, null);
                        ReceivedTemp = "";
                    }
                }
        }
        catch (TimeoutException)
        { }
```

```
        catch (Exception ex)
        {
            throw;
        }

    }

    public static void Send(string data)
    {
        setLog("Send: " + data);
        sport.Write(data);
    }

    private static void serialport_connect(String port, int baudrate, Parity parity,
int databits, StopBits stopbits)
    {
        sport = new SerialPort(port, baudrate, parity, databits, stopbits);
        try
        {

            sport.ReadTimeout = 1000;
            sport.WriteTimeout = 1000;
            sport.Open();

        }
        catch (Exception ex)
        {
            setLog(ex.Message);
            throw;
        }
    }

    public static void sport_DataReceived(object sender, SerialDataReceivedEventArgs
e)
    {
        string input = (string)sender;//sport.ReadExisting();
        setLog("Received: " + input);

        DataReceived(input);

        string[] data = input.Split(':');
        switch (data[0])
        {
            case "L0": Light[0] = double.Parse(data[1]); break;
            case "L1": Light[0] = double.Parse(data[1]); break;
            case "L2": Light[0] = double.Parse(data[1]); break;
            case "L3": Light[0] = double.Parse(data[1]); break;
            case "L4": Light[0] = double.Parse(data[1]); break;

            case "S0": Light[0] = double.Parse(data[1]); break;
            case "S1": Light[0] = double.Parse(data[1]); break;
            default: info = input; break;
        }
    }

    public static double getServoAngle(int ID)
    {
        return ServoAngle[ID];
    }

    public static double getLight(int ID)
    {
        return Light[ID];
    }
```

```
    public static string getInfo(int ID)
    {
        return info;
    }

}
```

CD

GITHUB-LINK: https://github.com/markxma1/Phototropi