4/5/2022

Design document

Blueprint Automation

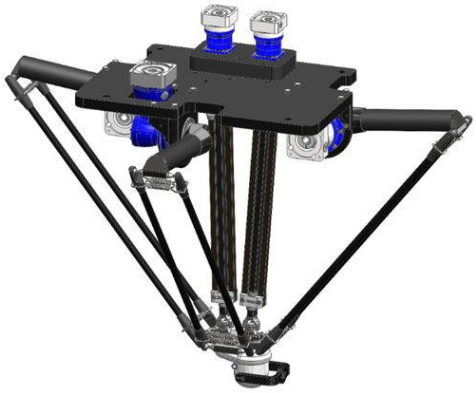
Hidde Roos, Julian de Jong, Christian Diekmann & Sanne Hermans

Contents

The concept

This project aims to create a JavaScript application that simulates the movement of a delta robot in real-time. To do achieve a live simulation of the robot, a reliable communication between the PLC controlling the robot and the JavaScript application is essential. The project also plans to store the retrieved simulation data in a database that will allow a replay of the simulation at a later time.

System description



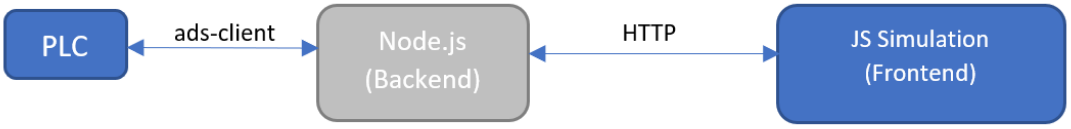
The system of the project is divided into 3 main parts:

- The physical delta robot: This is the robot being simulated in the JS visualisation application. The delta robot has 4 actuators which values are needed for the simulation. The delta robot is controlled by a Beckhoff PLC, so we will view them as one system.
- The Node.js application: This will be the backbone of the project. The application will connect to the Beckhoff ADS data layer of the PLC to read real-time data of the actuators. It will also be hosting the actual visualization on a webserver and server the simulation with the actuator's values.
- The JavaScript application: The JavaScript application will be the simulation of the delta robot. The JavaScript will communicate with the Backend to retrieve the correct data.

Context diagram

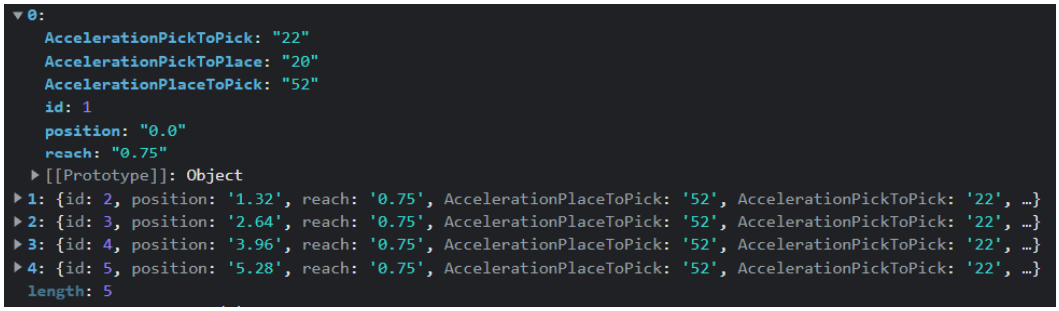
Component	Responsibility's	Language and system
PLC	The PLC controls the real delta robot(s) and makes the control data available through the ADS interface router.	Structured Text, Beckhoff
Backend Application	The backend application will communicate over the PLCs message router via a TCP/IP connection to request the control data in real-time.	Node.js
Frontend Application	The frontend application will display the robot simulation and communicate with the backend with a general web API.	JavaScript and HTML

A basic overview of the communication route between the PLC and the simulation in the browser is illustrated below.



System configuration

The configuration data is stored in an XML file format on the schedular (external to the PLC). It contains information about the number and type of robots, as well as other parameters of the system. For our project we have the configuration file stored on the webserver in the backend. The file will be requested by the frontend directly after a connection is established and load the correct number of delta robots into the environment. The image below shows the array of robot objects created from the xml configuration file.



Database

The database will be filled with the data we send to the application. This will be in Json format. The database will be an USB-stick we put in the PLC. All the last data will be in this database. When the USB is full, we delete the oldest data end place the new data in it.

The Json will look like this:

{

“Motor1”: “90”,

“Motor2”: “45”,

“Motor3”: “45”,

“Motor4”: “180”,

“Timestamp”: “05-04-2022 10:35:07”

}

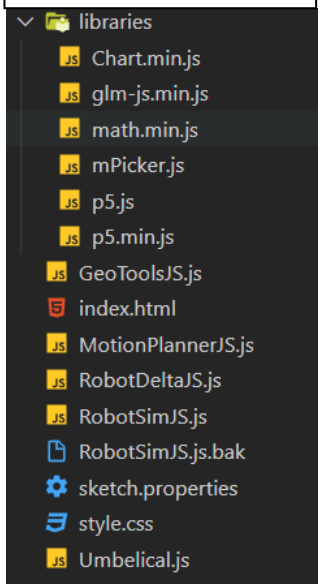
Data:	Meaning:
Motor1	Angular position of the first delta motor in degrees.
Motor2	Angular position of the second delta motor in degrees.
Motor3	Angular position of the third delta motor in degrees.
Motor4	Angular position of the fourth delta motor in degrees.
Timestamp	The time this data was measured.
Hash	To see which robot is changing

JavaScript Visualization

Refactoring example code

At the start of this project, we received some example code of a delta robot visualization. But this code was written in old JavaScript and is not object-oriented. To make sure this code can be re-used in future programs we decided to refactor it to an object-oriented standard.

Files in the example program



The example program

First, we had to understand the example code and its libraries.

The example code uses a couple of libraries:

- P5 to draw 3D objects
- mPicker to assign each object with an ID
- math for calculations
- glm-js for more math functions
- Chart to draw graphs

We will re-use these libraries in our refactored code.

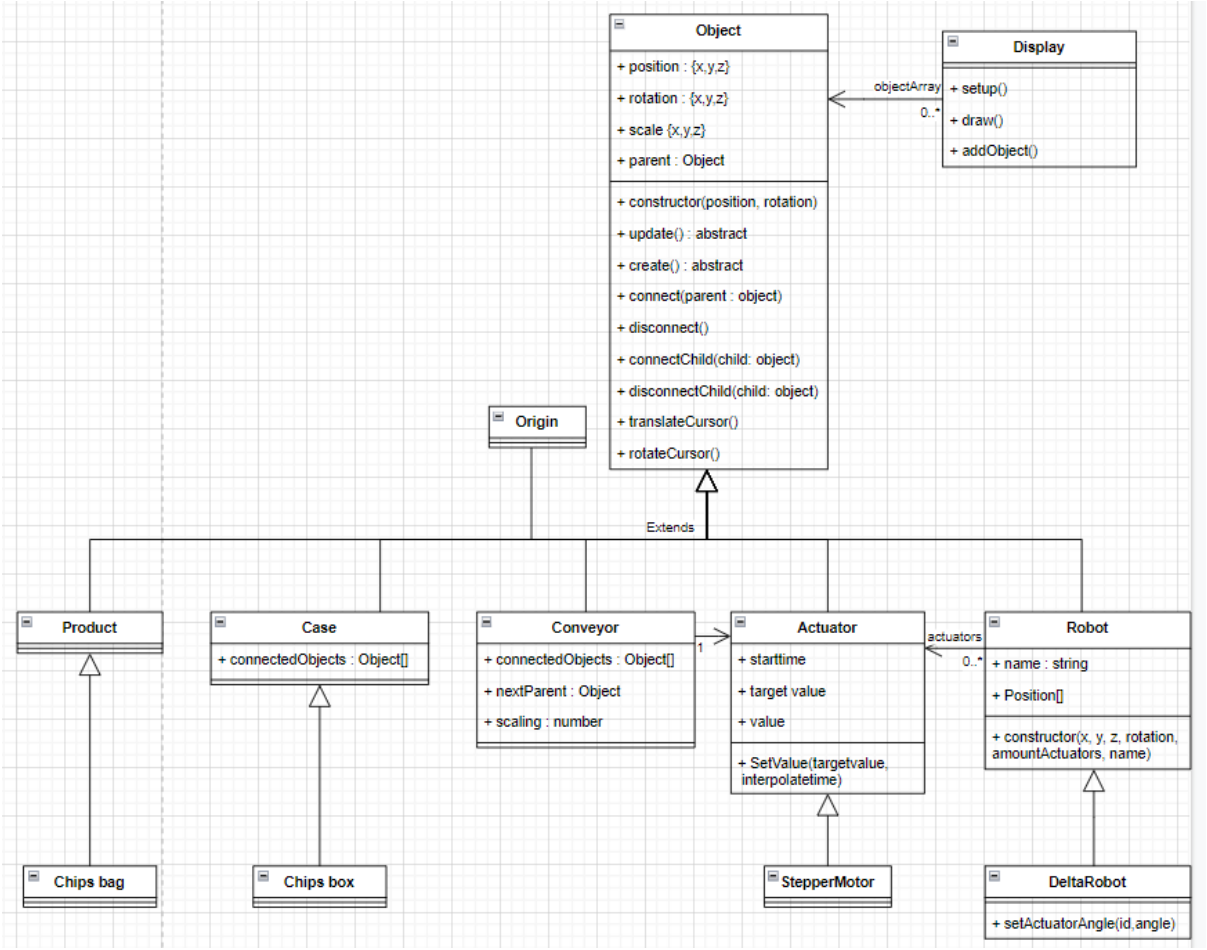
The example code has a couple of different files:

- RobotDelta.js : is used to define the delta robot that should be drawn.
- RobotSim.js : is used to draw the simulation and creates the RobotDelta .
- MotionPlanner.js : is used to plan the movement of the RobotDelta.
- Umbelical.js : is used like a library and has functions for mechanical calculations.
- GeoTools.js : is used like a library and has more math functions.
- PapaParse.js : is used for reading the CSV files

We will move the library files to a library folder in the refactored application and convert the RobotDelta and RobotSim to a more generic class so it can be re-used in other applications. We will not use the MotionPlanner in the refactored program as we plan to move the RobotDelta using its setActuatorAngle(id, angle) method.

Class Diagram refactored code

Before we started working on refactoring the code, we created a class diagram to make sure the design meets all requirements.



Class name	responsibility
Object	All classes that want to be displayed on the screen, inherit from this class
Display	Central class which Calls the Update() and Create() method on each added Object
Actuator	Superclass for actuators to inherit, stores the actuator's value
Robot	Superclass for Robots to inherit, contains an array of Actuators
Product	Superclass for products to inherit
Case	Superclass for cases to inherit, controls the position of its children to be inside the case
Conveyor	Superclass for conveyors to inherit, controls the position of its children to simulate them moving over the conveyor
Origin	Example Object class
Chips bag	Example Product class
Chips box	Example Case class
StepperMotor	Example Actuator class
DeltaRobot	Example Robot class

Super class ‘Object

All objects that should be drawn on the screen (like delta robots, servos, or products) have a couple of things in common: they all have a position in the world (x,y,z and rotation variables) so the program knows where to draw it, and they all have a method that tells the program what the object looks like (Create() method). To be able to move/ change the object over time, they all have an Update() method which is called before each Create() method is called.

To avoid having to copy this code in each new drawable object, we made a ‘super class’ called ‘Object’. Every class that should be drawn on the screen, inherits from Object so that they have all required methods and variables to be drawn on the screen.

methods and fields:

+ create() method

This method is called every frame and will create the object that should be displayed. This should be overloaded by other classes, as every class has a different model that should be displayed.

+ update() method

This method will be called before every create() method. This can be used to change the position of the object before it is displayed.

+ connect(parent) method

This method will assign a parent to this object. While a parent is assigned, the position of this object will be an offset position relative to its parent’s position. For example, this is used by conveyors to simulate the objects moving over the conveyor by changing the objects relative position according to the conveyor’s speed.

+ connectChild(child) method

This method is called when a child will assign this object as parent. Other classes (like Conveyor and Case) can override this method. For example: the Case class will use this to position the child inside the case when the connectChild(child) method is called.

+ disconnectChild(child) method

This method is called when a child will disconnect from this parent. Other classes can override and use this method to revert any changes that were made to the child by this parent.

+ translateCursor() method

When displaying models or objects with the p5 library, it places them at the current position of the p5 cursor. This method will move the cursor to the correct position, while also taking the parent’s position and rotation into account.

+ rotateCursor() method

This method will rotate the p5 cursor to the correct rotation, while also taking the parent’s rotation into account.

Central class ‘Display’

To draw each object to the screen, something must call each of their Update() and Create() methods, this is what the Display class does. The Display class has a list of Objects called ‘objectArray’. Each frame, the Display class will go through this list and call the Update() and Create() methods respectively. To add an object to this list, the Display class has an ‘addObject()’ method.

methods and fields:

+ addObject() method

Adds this object to the object list, so its update() and create() methods are called every frame.

Super class ‘Actuator’

Used to connect to an actual actuator of a machine and sync its value’s.

methods and fields:

+ value field

The value that belongs to this actuator, for example: a servo will store its rotation, or an encoder will store its pulsecount.

+ SetValue(targetValue, interpolatetime, callback, originalobject) method

This method will set a target value, the actuator will smoothly move to the target value.

- ‘Interpolatetime’ is how long in milliseconds it will take to get to the target value (can be set to 0 to immediately set the value).
- Callback: is an optional reference to a callback. It is called when the target value is reached.
- When using a callback, **‘originalObject’ should point to the object that contains the callback.** Because, in JavaScript, the ‘this’ field does not always point to the same object when called from a different class or from a callback. So, the originalObject should be passed to keep a reference to the correct object.

Super class ‘Conveyor’

Places all connected child objects on top of the conveyor and moves them along its length.

methods and fields:

+ connectedObjects field

This field contains a list of connected objects. This is used in the Update() method to move these objects to the correct position on the conveyor.

+ nextParent field

This field is a reference to the next parent. When the objects on the conveyor reach the end of the conveyor, they will be connected to the next parent if there is one.

This can be used to chain conveyors together, or to put the products in a box at the end of the conveyor.

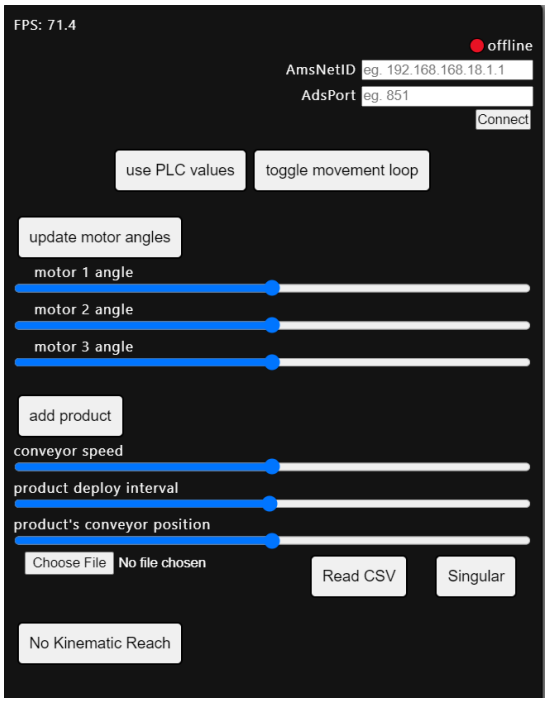
+ scaling field

Is used to translate the connected actuator's value to a distance in the visualization.

Super class 'Case'

Overrides the ConnectChild(child) method from displayObject to control the connected child's position to place it inside the case.

Use The Program



On the right side of the program we added items in the hud for testing. The top part of the hud is for connecting with the PLC and testing the PLC data.

Underneath the PLC info there is a button with three sliders to update the angles of one of the deltarobots for testingpurposes.

Below that there are options for the conveyor belt. You can change the speed of the conveyor belt. Add products change the deployment interval and change the placement position of the objects on the conveyor belt.

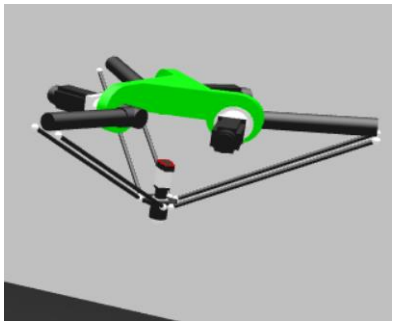
At the bottom of the hud there is an option to read a CSV file that needs to be created by the PLC. The idea is that you can choose the file from the usbstick from the PLC and read the data out. It is possible to only read the data once or repeat reading the csv file. If the data is reading on repeat. You can stop reading the data by changing the button to Singular.

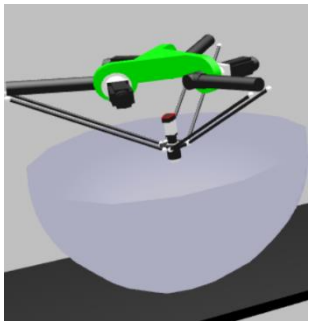
The CSV file checks for the correct hash to know if it is the correct file to read for the specific object.

The DeltaRobot –Demo  file is a working file to test the CSV reader.

DeltaRobot - Demo.csv

At last there is a button to enable or disable the reach of the deltarobot.

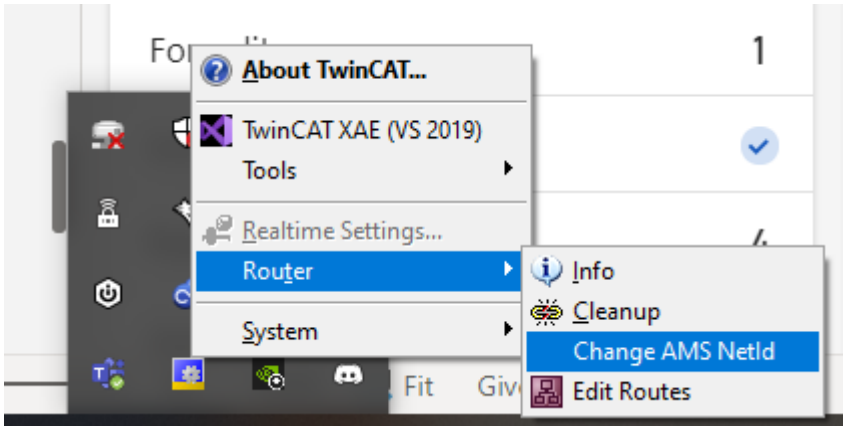




Javascript Backend

Connecting

To start the server code, run the command “npm start” in the console. When you do that, the program starts to run, and it connects with the front-end. Then you type in the IP address in the front-end that you want to connect to. You can find the correct Ip address when you go to change AMS NetId. You don't must change it but you can copy the current Ip address.



Then you type in the port you want (the port is always 851) and you press connect. If everything is good the red dot will become green, and you see that it is connected. You can also see the current state of the PLC program.

Explanation of the code

Node.js (Backend) --> PLC

The communication between the PLC and the backend application will make use of the unofficial ads-client library. A complete documentation of the client is available here. A brief description of the functions used by our application is given in the table below.

Function:	Description:	Expected response:
connect()	Connects the client to the PLC according to the pre-defined AmsNetID and AdsPort	Returns a promise, if resolved a connection was established successfully. If rejected an error is returned and logged to the server log database.
connection.connected	Checks whether there is an active connection to the PLC	Return true when connected, else false
readPlcRuntimeState()	Reads the runtime state of the PLC	Returns a promise, if resolved the status is returned. If rejected an error is returned and logged to the server log database.
readSymbol(“<string>”)	Reads a variable from the PLC	Returns a promise, if resolved, the variable value and data type are returned. If rejected an error is returned and logged to the server database.
disconnect()	Disconnects the client from the PLC and unregisters ADS port from router	Returns a resolved promise if disconnect was successful else

		an error that will be logged. The client will disconnect, regardless.

Node.js <--> JS Simulation

Currently all communication between the front and backend application is handled by asynchronous fetch() requests. An attempt to use a WebSocket for sending the actuator positions was made but has not been implemented yet.

Route:	Description:	Expected response:
`'/connect'`	Creates an ads client object with the provided AmsNetID and Ads port and initiates a connection.	200 on successful connection or 500 on unsuccessful connection
`'/config'`	Requests the xml system configuration file	An xml file containing the system configuration
`'/connected'`	Checks the client's connection status	200 if connected and 500 if disconnected
`'/state'`	Checks the PLC's runtime state	A Json containing the runtime state as Enum and string
`'/pos0'` `'/pos1'` `'/pos2'`	Reads the respective actuator position variables from the PLC	A Json containing the variable value and data type
`'/disconnect`	Disconnects the client from the PLC	200 on clean disconnect or 500 if disconnected with errors

Database

All errors encountered by the server are logged into a database file that is stored along the server.js file.

PLC code

The sample programme running on the PLC uses the system time as the seed for the DRAND function block to generate three random numbers in the range of 0 to –1. These numbers are mapped to a respective stepper motor and only used for testing the simulation. All variables are in the GVL and therefore in the global scope as well as the ADS layer. Two TON timers are used to generate new random values at an interval. Changing the timers value changes the interval at which new numbers are generated.

Save to USB

We are saving the data to a csv file on a USB drive so we can replay what the delta robot has done. This program is not really finished but it saves one item in the CSV file. When you want to run this program you fhirst have to make a CSV file in ‘C:\Temp\TextModeGen.csv’ you can also add a CSV file somerwhere else but then you have to change the path in the code. The code you have to look at is in this file. If you have the file you can run the code. To make the code run you have to change the bWrite value to true like I did here. when you press the ’write values’ button the value changes. When you look in the CSV file the first row will be changed.