MTConnectVR Documentation

Chris Sculley <[cs.sculley@gmail.com](mailto:cs.sculley@gmail.com)>

# Scripting Overview

At the top of the script, I split up the variables into three main sections: Properties, Settings, and References. I use the C# attribute [Header(“HeaderName”)] above the first variable to separate the variables in the Unity Inspector.

Properties are public variables (and will appear in the Inspector) that represent the state of the script. Ideally, these can be changed by external scripts and nothing will break. An example of this is below:

[Header(“Properties”)]

public float currentHeight;

If you need to run functions on getting or setting, consider using a private variable with C# automatic getters and setters. An example of this is below:

[SerializeField] private float \_currentHeight;

public float CurrentHeight {

get { return \_currentHeight; }

protected set { \_currentHeight = value; }

}

A benefit of this is that you can easily specify privilege levels based on get and set. The [SerializeField] attribute before the private variable will make the variable appear in the Unity editor, so you can still see it’s value. Be aware that changes in the editor will not trigger the getter/setters though!

Settings are variables that are intended to be set in the inspector before hitting the play button. These values are often used for initialization and should only change under very rare circumstances. An example of this is below:

[Header(“Settings”)]

public float startingHeight = 0.5f;

Finally, References are references to other instances in the Scene. These have to be populated in the inspector by dragging the needed GameObject or Component into the box in the right, so you have to check for null values! An example of this is below, with error checks happening in Awake().

[Header(“References”)]

public GameObject sphere;

public Kuka kukaRobot;

private void Awake() {

// Safety checks

Debug.Assert(sphere != null, “Reference to sphere in scene is null!”);

Debug.Assert(kukaRobot != null, “Reference to GameObject with Kuka script is null!”);

}

Based on the order of execution, I perform initialization and error checking based on Awake(), OnEnable() and Start(). Unity will call Awake() and OnEnable() back-to-back for every script attached to a GameObject in the scene first, and then call Start() for every script. Thus, I perform all basic error checking in Awake() and adding of any static references if the script is a singleton. However, I do not check if any static references are valid until Start(), where I generally perform script initialization and error checking of static references. Thus, every other function except Awake() can check or use static references.

# Singletons

Currently there is only one singleton, which is MTConnect, and machines such as the Kuka and Shark try to add themselves to the current MTConnect instance when they are initialized. I plan on changing MTConnect to just be an input source and creating a new singleton that will handle adding and removing machines from a global List<Machines>, and new input sources.

* MachineManager.cs – Coming soon 😊
* MTConnect.cs – Gets parsed data from MTConnect data stream (<http://128.46.131.12:8000/>). Will change to parse raw MTConnect XML (<http://128.46.131.12/>) and send data to singleton.
  + Static References: mtc (MTConnect)
    - MTC: Static reference to self. Is populated in Awake().
  + Properties: machines (List<Machine>)
    - Machines: Dynamic list of machines. Currently, shark is hardcoded as machines[0], and Kuka is hardcoded as machines[1].
  + Settings: MTConnectURL (string), pollInterval (float), delim (char), trimChars (char[])
    - MTConnectURL: Base string with URL to parse from. Should be <http://128.46.131.12/>.
    - PollInterval: Interval in seconds to pull new data from internet. Default value is 0.1 seconds, or 10 hz.
    - Delim: Delimiter character. Should be ‘,’ for data stream.
    - TrimChars: Array of chars to trim from either side before splitting array. Should be ‘”’ and ‘,’.
  + Debug: debugAxes (float[]), useDebugString (bool), useOpenHaptics (bool)
    - DebugAxes: Array of 9 elements. Set this in the inspector. If “useDebugString” is true, this will override the data from the data stream (<http://128.46.131.12:8000/>).
    - UseDebugString: If this is true, than instead of using the live values retrieved from the data stream, will use the values in DebugAxes instead.
    - UseOpenHaptics: Used to disable MTConnect polling. Set to true to disable polling.
  + Public Methods: void AddMachine(Machine m)
    - AddMachine (Machine m): Adds a machine to the machines List<Machine>
  + Private Methods: IEnumerator FetchMTConnect()
    - FetchMTConnect(): Fetches the values at the URL and parses the angles. Sets the Shark and Kuka angles through machines List<Machine> directly for now.

# Machines

Machines inherit the base class Machine, and then add on functionality. The base class needs to be updated to allow for rotary or linear axes, currently assumes all axes are rotary. Will probably change other fields to match MTConnect’s format a bit more, too.

* Machine.cs – Base class all machines inherit from.
  + Machine Properties: angles (float[])
    - Angles: Array of floats with values for each axis. Will probably change to “axis” to be more general.
  + Machine Settings: axisCount (int), maxSpeed (float), manufacturer (string), model (string), name (string).
    - AxisCount: Number of axis for machine
    - MaxSpeed: Maximum speed of machine for reference purposes
    - Manufacturer: String with manufacturer
    - Model: String with model
    - Name: Name of machine
  + Public Methods: Vector3 GetAxis(int axisID), SetAxisAngle(string axisName, float angle)
    - GetAxis(int axisID): Returns a vector3 with the local space axis of rotation. Only valid for rotary axis.
    - SetAxisAngle(string axisName, float angle): Sets the axis angle based on the name of the axis. Each machine is in charge of defining the name for each of its own axis.
* Shark.cs – Machine script for Shark CNC router
  + Shark Settings: lerpSpeed (float), interpolation (bool), minAngles (float[]), maxAngles(float[])
    - LerpSpeed: Speed to lerp to “correct” position. Will change to linear speed-based interpolation instead of exponential-based interpolation that it is now.
    - Interpolation: Boolean that controls whether the Shark will gradually interpolate to the values set in angles[] or instantly jump to correct values.
    - MinAngles: Sets the minimum value for each corresponding angles[] value. If both min and max angles are 0, no restriction will be used (can move freely).
    - MaxAngles: Sets the maximum value for each corresponding angles value. If both min and max angles are 0, no restriction will be used (can move freely).
* Kuka.cs – Machine script for Kuka
  + Kuka Settings: lerpSpeed (float), interpolation (bool), minAngles (float[]), maxAngles(float[]), samplingDistance (float), learningRate (float)
    - LerpSpeed: Speed to lerp to “correct” position. Will change to linear speed-based interpolation instead of exponential-based interpolation that it is now.
    - Interpolation: Boolean that controls whether the Kuka will gradually interpolate to the values set in angles[] or instantly jump to correct values.
    - MinAngles: Sets the minimum value for each corresponding angles[] value. If both min and max angles are 0, no restriction will be used (can move freely).
    - MaxAngles: Sets the maximum value for each corresponding angles value. If both min and max angles are 0, no restriction will be used (can move freely).
    - SamplingDistance: Controls the distance used to generate the partial gradient for each axis. Smaller is more precise.
    - LearningRate: Learning rate of gradient descent, higher values will cause the Kuka to perform inverse kinematics faster, but can cause irregularities or jumping around the final point if too high.
  + Public Methods: Vector3 ForwardKinematics(float[] anglesToCalculate), void InverseKinematics(Vector3 target)
    - ForwardKinematics(float[] anglesToCalculate): Returns the location of the tip of the Kuka in world space based on the corresponding passed angles. Used to perform gradient descent for inverse kinematics.
    - InverseKinematics(Vector3 target): When called, moves the Kuka slightly towards the passed target in world space.
  + Private Methods: NormalizeAngles(float angle), PartialGradient(Vector3 target, float[] anglesToCalculate, int axisID)
    - NormalizeAngles(float angle): Normalize angles between 0-360 degrees. Not using this function can cause weird calculations when not using Quaternions.
    - PartialGradient(Vector3 target, float[] anglesToCalculate, int axisID): Internal function used for Inverse Kinematics. Returns the gradient for the specific angle passed as “axisID”.

# Other Scripts

Other scripts in the scene.

* Webcam.cs – Used to get webcam data from local webcam. Displays the webcam feed onto whatever object it is attached to, and turns to face the main camera.
* OpenHapticsConnect: Will be used to connect to the PHANTOM haptics device.
* Test.cs: Calls InverseKinematics every frame on Kuka with current location as the target location. Will be removed soon 😊