Tracking4All Documentation

Early Access V1.02

Contents

Introduction

Overview

Tracking4All is a modular and flexible framework that can be used to power the body tracking features of games/projects. It was created through game development experience publishing real projects and focuses on modularity and practicality in the context of project prototyping and production. It aims to be backend independent meaning that the features and solutions in the framework should work with a variety of backend tracking/detection approaches varying from Mediapipe to custom solutions. This makes the framework future proof as well as very flexible for use in a variety of cases.

Key Features

- Support for; pose tracking, and hand tracking, face tracking (coming soon).
- Independent to backend solution (can be powered by solutions ranging from Python server to embedded C++ solutions).
- Cross-platform (Windows, Mac, Android, iOS).
- Aims to be powered by hardware that is accessible to end-users (webcam with no unreasonable setup requirements).
- Modular Features: can optionally add features that extend on the core framework (ex: avatar puppeteering, landmark modifiers, avatar face tracking, etc.) – again independent to the backend data provider.

Getting Started

Quick Start Guide

The fastest way to get started is to open one of the example scenes at 'Tracking4All/Scenes'. Running play will automatically connect to your default webcam and show the solution running.

Either work in a duplication of these examples scenes or copy the necessary objects (Solution, Tracking4All, EventSystem) are usually required at minimum.

It is highly recommended to read and understand the 'Core Concepts' section further below for most projects.

Setting up Your Project

It is highly recommended to use the provided project files as it includes special player settings and configurations to allow the framework to function and build correctly.

If absolutely required, you can copy the assets into your custom unity project. You may need to adjust player settings and other files to ensure the framework builds correctly.

Core Concepts

Architecture Overview

The core of Tracking4All can be thought to conceptually consist of 3 layers:

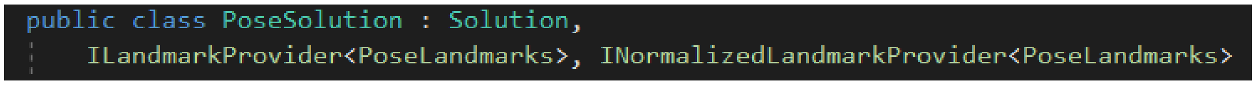
1. Detection Layer: Something needs to provide detection data to Tracking4All (e.g. Landmark data, etc.). Tracking4All doesn't care what the Detection Layer is or how it's powered – however the layer must provide at least the necessary data to power the solutions you want to use. Tracking4All ships with an embedded detection layer (C++) and a simple server-based detection layer (Python).
2. Adapter Layer: The Adapter Layer is responsible for parsing data from the Detection Layer and converting it into data that Tracking4All understands.
3. Solutions Layer: The Solutions Layer exposes data that can be read/used in a friendly manner by you and other plug-ins. This layer is easy to use and guaranteed to be Unity thread safe when used properly.

Additional non-essential layers:

- Modifier Layer: The Modifier Layer post processes data on the Solutions Layer (e.g. smoothing, transformations, etc.) Modifiers can generically act on as many types of data as the modifiers are implemented to support.

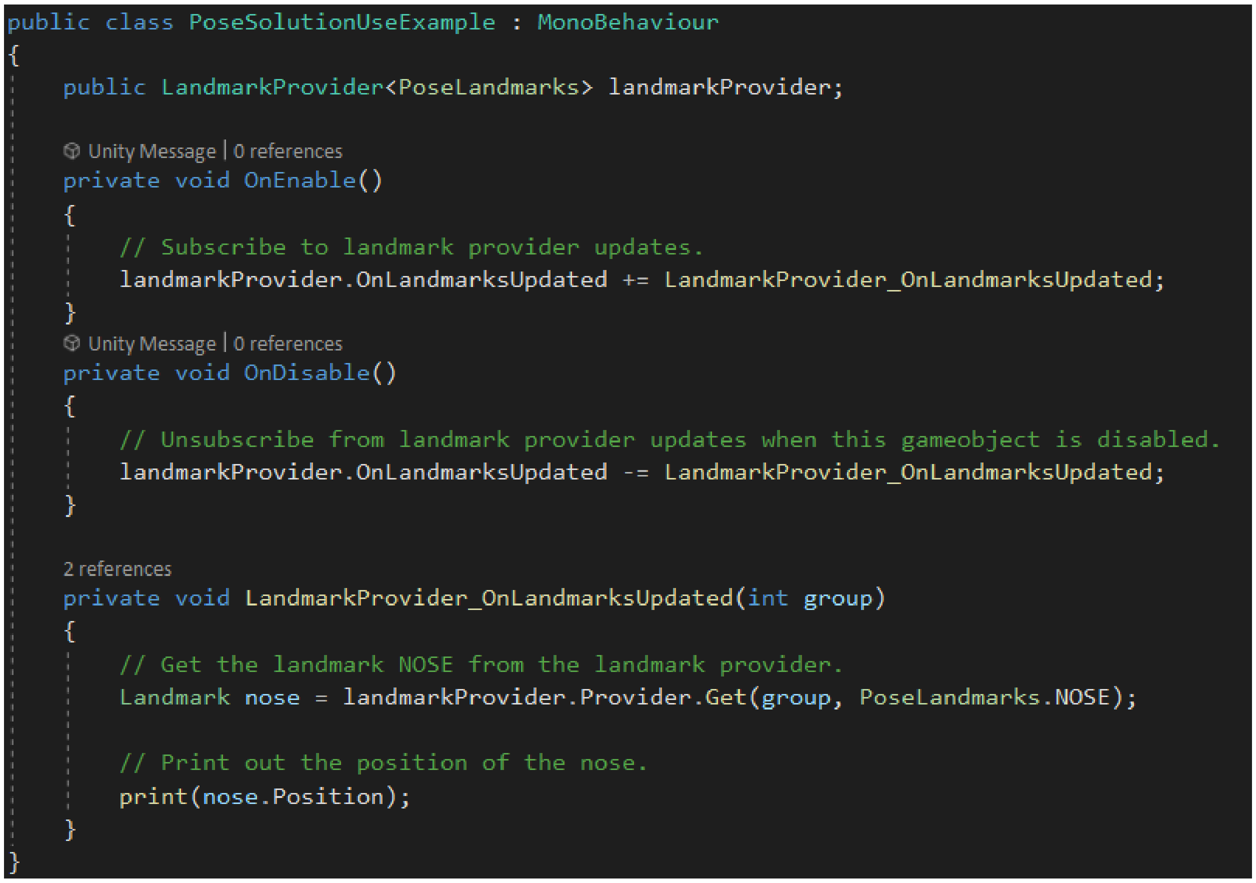
Basic Workflow

1. Solutions are premade for you in the 'Tracking4All/Scenes' folder. It is highly recommended to use the solutions here to begin with.
2. Once you have decided on which solution to use (e.g. Pose). Now decide which solution provider you will use:
    - Just answer the question: do I want detection to run on the target device or remotely? Tracking4All ships with an embedded detection layer (C++) and a basic server-based detection layer (Python).
    - You can choose which detection layer is powering the framework by enabling/disabling the appropriate game object starting with 'MPU' (Mediapipe Unity) or 'MPP' (Mediapipe Python) and then updating the Adapter's provider fields using the unity inspector.
3. You read data from Solutions using Providers. You can check what type of data Solutions can provide by inspecting the Solution class definition. For example, check the Pose Solution:



```
public class PoseSolution : Solution,
    ILandmarkProvider<PoseLandmarks>, INormalizedLandmarkProvider<PoseLandmarks>
```

You can see that this PoseSolution can provide Landmark and NormalizedLandmark data. Here is a very basic example of how to access data from this PoseSolution:



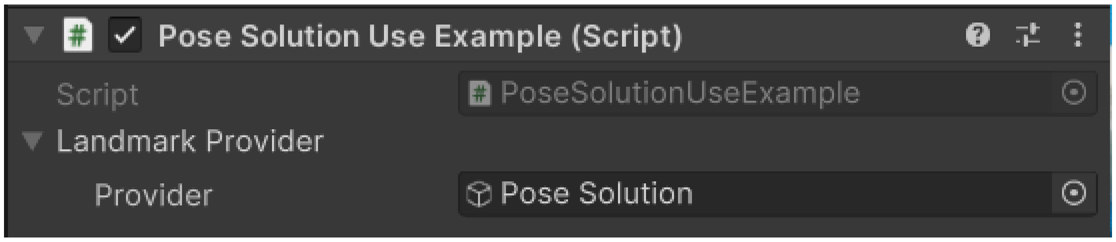
```
public class PoseSolutionUseExample : MonoBehaviour
{
    public LandmarkProvider<PoseLandmarks> landmarkProvider;

    Unity Message | 0 references
    private void OnEnable()
    {
        // Subscribe to landmark provider updates.
        landmarkProvider.OnLandmarksUpdated += LandmarkProvider_OnLandmarksUpdated;
    }
    Unity Message | 0 references
    private void OnDisable()
    {
        // Unsubscribe from landmark provider updates when this gameobject is disabled.
        landmarkProvider.OnLandmarksUpdated -= LandmarkProvider_OnLandmarksUpdated;
    }

    2 references
    private void LandmarkProvider_OnLandmarksUpdated(int group)
    {
        // Get the landmark NOSE from the landmark provider.
        Landmark nose = landmarkProvider.Provider.Get(group, PoseLandmarks.NOSE);

        // Print out the position of the nose.
        print(nose.Position);
    }
}
```

Then create a new gameobject, add the new component, and assign the gameobject which contains the PoseSolution in the inspector like so:



Pose Solution Use Example (Script)

Script: PoseSolutionUseExample

Landmark Provider

Provider: Pose Solution

After pressing play, the position of the nose will be output in the console.

Core Datatypes

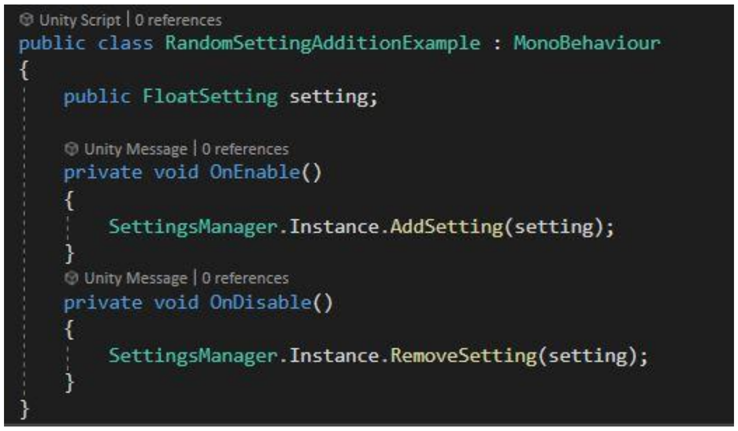
Landmark: contains 3D coordinates from the detection backend.

NormalizedLandmark: coordinates of a landmark in camera screen space in a [0,1] range. XY coordinates [0,0] is the bottom left of the camera view and [1,1] is the top right of the camera view. The Z-component is the approximate depth of the landmark.

Framework Guides

Adding New Runtime Settings

It is very easy to add new runtime settings which can be modified in the menu. For example, I can add a new float setting to the menu with this class in the scene:



```
Unity Script | 0 references
public class RandomSettingAdditionExample : MonoBehaviour
{
    public FloatSetting setting;

    Unity Message | 0 references
    private void OnEnable()
    {
        SettingsManager.Instance.AddSetting(setting);
    }
    Unity Message | 0 references
    private void OnDisable()
    {
        SettingsManager.Instance.RemoveSetting(setting);
    }
}
```

- You can subscribe to the ValueChangedInMenu delegate to listen to menu caused changes of your setting.
- You can read and set the setting using setting.Value and seting.Set().

Experimental Guides

Avatars/Puppeteering

Puppeteering arbitrary humanoid avatars is supported however is a little bit rough around the edges currently. This will improve over time. Tracking4All generates full transform data given a pose data provider by driving a Puppet. Avatars then retrieve the transform data from a puppet to drive the motion of the avatar.
Features:

- Supports arbitrary tracked rotations (you can turn around, do a head stand, etc.).
- Supports most humanoid avatars which are correctly imported as humanoid via Unity Mecanim.

The quickest way to experiment with puppeteering avatars is by previewing the Avatar example scene. You can control Unity-chan in this demo scene and examine how it is setup or see further below for instructions on how to add your own avatar.

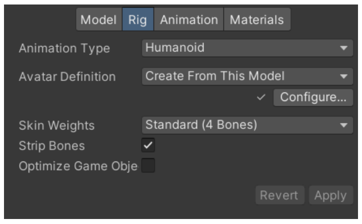
Notes:

- You can enable/disable the Visuals child of a puppet to visualize the puppet.
- At the moment ensure that the 3d model in the view starts aligned with the global world axes (temporary constraint). After loading, you can move the model however you'd like.

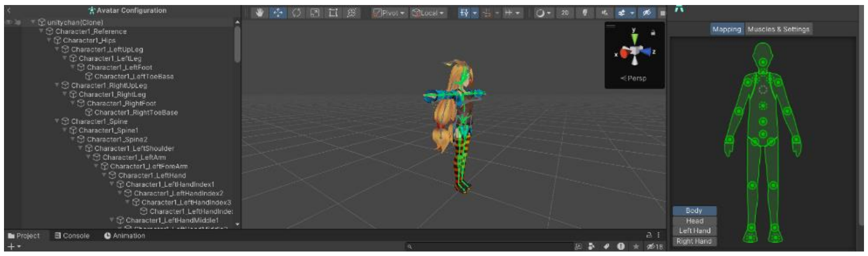
- Unity-chan is made available under the following license: https://unity-chan.com/contents/license_en/

Adding a New Avatar

1. Import the humanoid character into Unity.
2. Under the Rig import settings, ensure it is set to Humanoid and that the Avatar Definition is created.
3. Verify under Configure that there are no errors or warnings detected.
4. When in the Avatar Configuration window, it is recommended to copy the model in this window for pasting into your game scene since it is enforced in a good T-Pose (makes hooking up to the Puppet easier).
5. Paste the model into the scene you want the avatar in.
6. Add an Avatar component on the pasted model.
7. Assign the Pose Joint Provider to the Puppet in the scene and assign the Animator to the model's animator.
8. Play! That's it!



Animation Type	Humanoid
Avatar Definition	Create From This Model
Skin Weights	Standard (4 Bones)
Strip Bones	✓
Optimize Game Obje	



Avatar 3D Model Recommendations

The Avatar script can support a wide range of humanoids out-of-the-box. However, there are some guidelines to ensure quality tracking.

- Make sure the core bones are assigned in the Avatar configuration window: Hips, Spine, Chest, Neck, Head, Shoulders, Upper Arms, Lower Arms, Hands, Upper Legs, Lower Legs, Feet.
- The hips should ideally be the root of all the core bones. See Unity-chan's parenting structure. Is by chance your model has a different parenting structure, it is usually straight forward to adjust this in software such as Blender as necessary.

Creating Avatars at Runtime

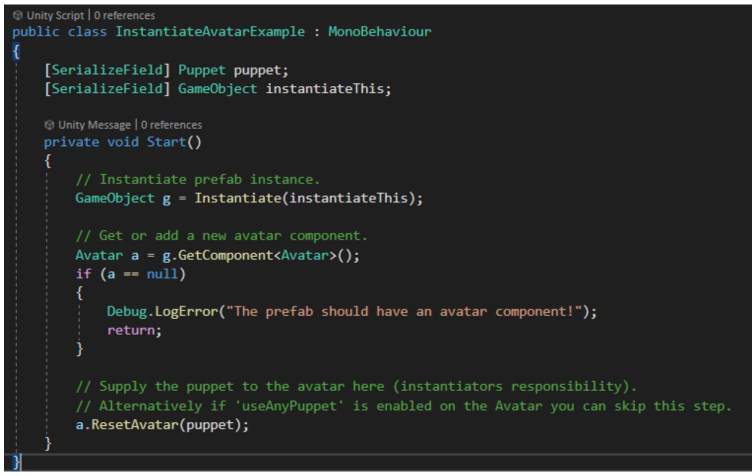
Creating avatars at runtime is supported. It is strongly recommended to use a prefab based approach rather than adding components at runtime.

1. Configure your 3D model correctly and add the Avatar script.
2. Assign all the required fields and test that the avatar is working as expected in the example scene.

3. Create a prefab out of this avatar.

When you want to instantiate a new instance of this prefab the primary issue is that the Puppet may not be a part of the prefab (so the avatar would spawn with a missing provider). The following approach is recommended in this case:

1. Instantiate the prefab in world space without any transformations.
2. Get the Avatar component and call Avatar.Reset(), supply the joint provider/puppet as an argument. Alternatively ensure the Avatar prefab has 'Use Any Puppet' enabled to bind arbitrarily to the first puppet found (not recommended).
3. The avatar should be functioning correctly.



```
Unity Script | 0 references
public class InstantiateAvatarExample : MonoBehaviour
{
    [SerializeField] Puppet puppet;
    [SerializeField] GameObject instantiateThis;

    Unity Message | 0 references
    private void Start()
    {
        // Instantiate prefab instance.
        GameObject g = Instantiate(instantiateThis);

        // Get or add a new avatar component.
        Avatar a = g.GetComponent<Avatar>();
        if (a == null)
        {
            Debug.LogError("The prefab should have an avatar component!");
            return;
        }

        // Supply the puppet to the avatar here (instantiators responsibility).
        // Alternatively if 'useAnyPuppet' is enabled on the Avatar you can skip this step.
        a.ResetAvatar(puppet);
    }
}
```

NOTE: the type of puppet can also be a JointProvider<PoseJoints>. Just be careful about carelessly passing a JointProvider to ResetAvatar, see the advanced usage section about IProviders.

Advanced Usage

In advanced use cases you can inherit from the classes used to power Tracking4All and implement custom functionality and additional data you want to support while also remaining compatible with most Tracking4All features.

You can also reference the source code explicitly to build custom Solutions, Adapters, and Detection layers.

IProvider and InterfaceProvider Peculiarities

This applies if you are using Providers and InterfaceProviders in your code (ex: LandmarkProvider, JointProvider, etc.) AND are doing checks to see if interfaces are null. If you want to avoid these peculiarities, just use concrete classes instead.

The IProvider of a serialized interface provider (ex: LandmarkProvider) will NEVER evaluate as null.

- You can **ONLY** reliably check if an interface provider has an underlying null interface by checking InterfaceProvider.HasInterface or InterfaceProvider.Null.

- Be especially careful when implicitly casting an InterfaceProvider to an interface and then checking if that interface is null (it never will be!)

Solution: if you want to treat an IProvider derived from a serialized InterfaceProvider as nullable use the Helpers.Nullable() extension method, it will return null or the IProvider correctly. Feel free to contact if there is any confusion.

Performance Tips

Adjusting the model settings depending on the detection layer will substantially affect performance.

You can disable the Visualizer components which display the solution output, data will still be updated as expected.

Support

Getting Support

Write to support@g8gaming.ca

Consider joining the Discord: https://discord.gg/7qHpEg8vPH

License and Acknowledgements

License Information

You should have gotten a LICENSE file in your assets folder.

Credits and Acknowledgements

ganthefans! Discord community for their advice and suggestions.

Homuler MediaPipe Unity Plugin: is used to provide the embedded Unity support.