

# Pulse Unity Asset User Manual

v3.0 - <https://goo.gl/GK1RZW>

## Table of contents

[Resources](#)

[Overview](#)

[Limitations](#)

[Unity components](#)

[Generating Data](#)

[PulseEngineDriver](#)

[PulseCSVReader](#)

[PulseRandomValueGenerator](#)

[Consuming vitals data](#)

[PulseDataLineRenderer](#)

[PulseDataNumberRenderer](#)

[Interacting with the patient](#)

[Examples](#)

[VitalsMonitor](#)

[CreateActionOnClick](#)

[ScreenSpace](#)

[RandomNumber](#)

## Resources

- Asset store page: [here](#)
- Discourse forum (support): [here](#)
- Issue tracker: [here](#)
- Pulse Physiology Engine documentation: [here](#)
- Pulse Unity Asset developer documentation: [here](#)

## Overview

The role of this asset is to integrate the Pulse Physiology Engine in Unity. It includes the following directories:

- **Data:** Data files necessary for the Pulse engine to create patients.
  - **Only if you plan on making new patients in game, will you need to copy this directory to a folder under the StreamingAssets directory located in the Assets folder of your project.**
- **Plugins:** Native (C-Style Interface to the C++ Pulse Engine) and managed (Pulse C# API and Google Protobuf) libraries to import into Unity. See the [developer documentation](#) for more information on how to update them manually.
- **Scenes:** Generic Vitals Monitor connected to a Pulse engine.
  - The Dev subfolder contains developer scenes that demonstrate the base functionality of the Scripts provided in this asset
- **Scripts:** [Components](#) and [scriptable objects](#) that can be used to conveniently interact with the Pulse Engine in your [scene](#).

The Pulse Unity Asset is based on the Pulse C# API. To learn more about working with the Pulse API, you can find HowTo folders in many of the [language folders in src](#) Pulse provides an API for each of these languages, and they are designed to be very consistent with each other. So if you are looking for how to use a particular feature in Pulse and the language you are using does not provide an example, look around in the other languages and it should be trivial to translate to the language of your application.

While advanced users could rely solely on the C# API of the PulseEngine imported in the plugins, we have created some useful components meant to facilitate interacting with Pulse in Unity.

*If you need help designing a custom solution based on Pulse or want to discuss collaboration, please contact us at [kitware@kitware.com](mailto:kitware@kitware.com).*

## Limitations

- **Missing functionality from the C++ framework:** The Pulse C# API should provide all [Common Data Mode](#) objects defined in the C++ framework. If you find any portion of the C++ missing that you need, let us know on our [Discourse](#) forum.

# Unity components

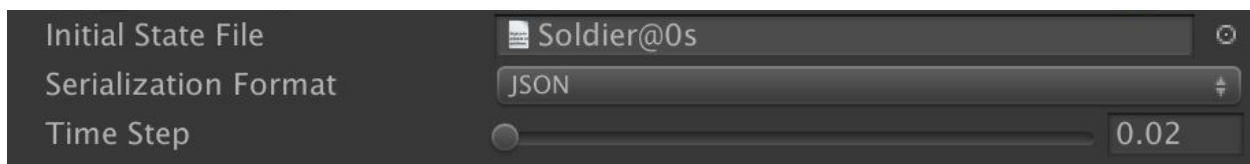
## 1. Generating Data

PulseDataSource is an abstract class which stores data organized per field in a PulseData scriptable object at each [MonoBehaviour.Update](#) and exposes it to be consumed by an instance of PulseDataConsumer. Its implemented subclasses are:

### PulseEngineDriver

This is the main component for a Pulse simulation as it creates an instance of PulseEngine and handles advancing the simulation time to maintain synchronization with the game clock. It exposes the PulseEngine so that other components can apply actions to it ([see below](#)), and populates PulseData with the vitals information generated by the engine, including simulation time, ECG signal, heart rate, arterial, systolic, and diastolic blood pressures, oxygen saturation, end tidal carbon dioxide, respiration rate, temperature, airway carbon dioxide partial pressure, and blood volume.

**Note:** This above list is only a subset of [the outputs available](#) in the C++ framework ([learn more](#)).



- **Initial State File:** initial stable state to use to start the simulation.
  - Pulse state files can be found in *Data > states*.
- **Serialization Format:** serialization format of the state file (JSON or BINARY).
- **Time Step:** simulation time step in seconds.

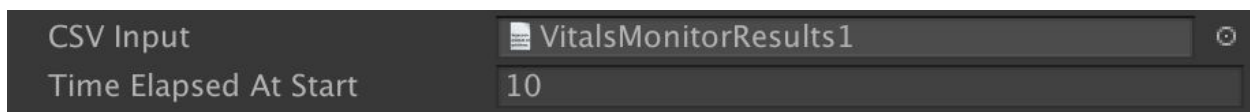
The PulseEngineDriver component encapsulates and provides access to the Pulse API C# PulseEngine object. This object is used to create, interact and retrieve data from an instance of a Pulse Engine. To access it, you could elect to either:

- subclass the PulseEngineController class designed for this effect (which also enforces a custom editor), or
- simply add a reference to a PulseEngineDriver in your custom component that defines when to apply an action.

To learn more about using the PulseEngine interface, start with understanding our [Common Data Model](#) and its [structures](#), as well as the [Pulse interface](#).

### PulseCSVReader

This component is used to display vital sign data stored in a comma separated value (csv) file from a previously executed pulse simulation.



- **CSV Input:** input TextAsset containing Pulse output data.. An example file can be found in Data.
- **Time Elapsed At Start:** offset time defining how many data points to read from the CSV file when the component is enabled.

### PulseRandomValueGenerator

This is a convenient component used to generate random values in a PulseData container.

Min Value	0
Max Value	100
Variability	0.2
Frequency	0

- **Min/Max Value:** range of the values being generated.
- **Variability:** fraction defining how much different the next generated value will be.
- **Frequency:** frequency at which values are generated

## 2. Consuming vitals data


PulseDataConsumer is an abstract class that listens to an instance of a PulseDataSource to consume its PulseData for a certain data field at each [MonoBehaviour.LateUpdate](#). Its subclasses (listed below) all inherit a custom editor with the following interface:

Data source	PulseEngineDriver (PulseEngineDriver)
Data field	Carina-CarbonDioxide-PartialPressure (mmHg)

- **Data source:** instance of the PulseDataSource to read the PulseData from.
- **Data field:** name of the data field to consume in the PulseData container.

### PulseDataLineRenderer

Renders a line (embeds a [LineRenderer](#)) that draws points based on the data field selected. One of its parents needs a [UI.Canvas](#) component with a Render Mode set to “Screen Space - Camera” to overlay it on the screen, or to “World Space” to place it on a 2D plane in the 3D scene. **Because the embedded LineRenderer is not part of the UI system, you can not set the canvas Render Mode to “Screen Space - Overlay”.**

Thickness	1
Color	
Trace Initial Line	<input checked="" type="checkbox"/>
Y Min	-15
Y Max	55
X Range	10

- **Thickness:** thickness of the line renderer
- **Color:** color of the line renderer
- **Trace Initial Line:** shows a flat line at time points where there are no read values yet
- **Y Min/Max:** range the Y axis representing the values of the data field selected

- **X Range:** range of the X axis in seconds. This axis is dynamic, with the far right being “now” (latest value), and the far left being “now - X Range”.

#### PulseDataNumberRenderer

Update the value of an associated [UI.Text](#) component to reflect the latest data field value. One of its parents needs a [UI.Canvas](#) component, and its Render Mode can be selected to World Space or Screen Space to respectively place it in the 3D world or overlay it on the screen.

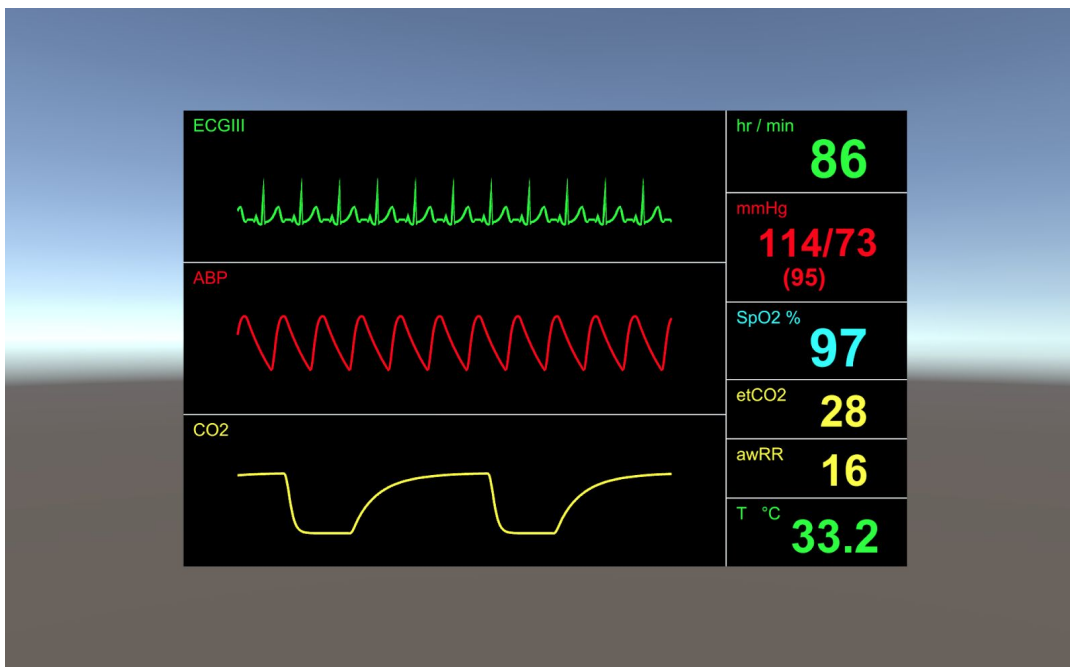
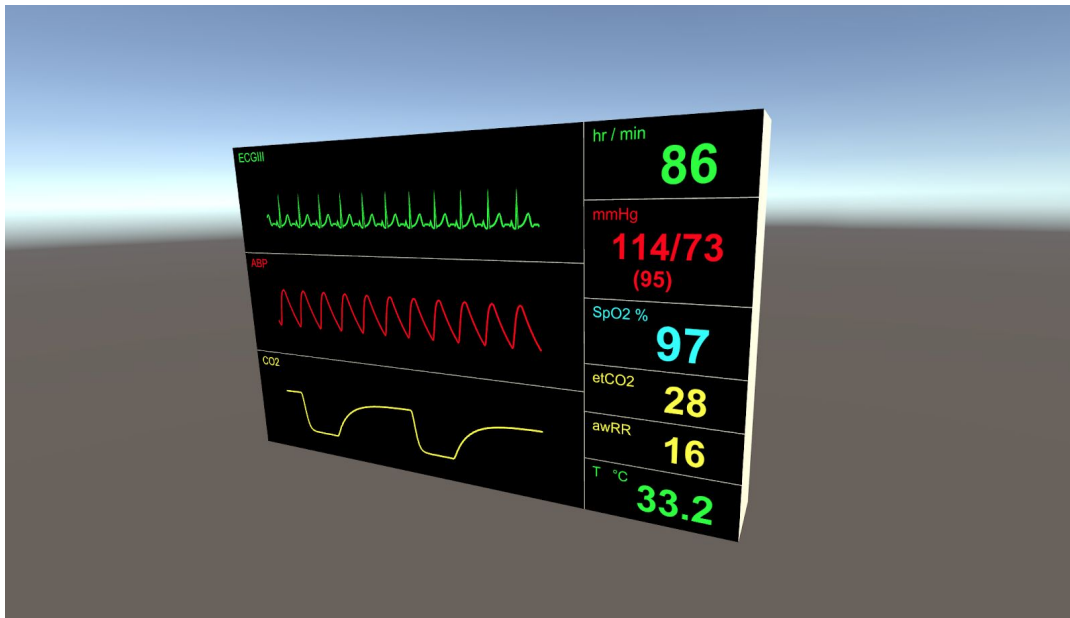
Prefix	Oxygen Saturation:
Suffix	%
Multiplier	100
Decimals	1
Frequency	<input type="range"/> 0

- **Prefix:** text prepended to the data value.
- **Suffix:** text appended to the data value.
- **Multiplier:** number multiplied to the value (useful to show percentages).
- **Decimals:** number of decimals shown.
- **Frequency:** frequency at which the number can be updated (0:no limit).

## Example Scenes

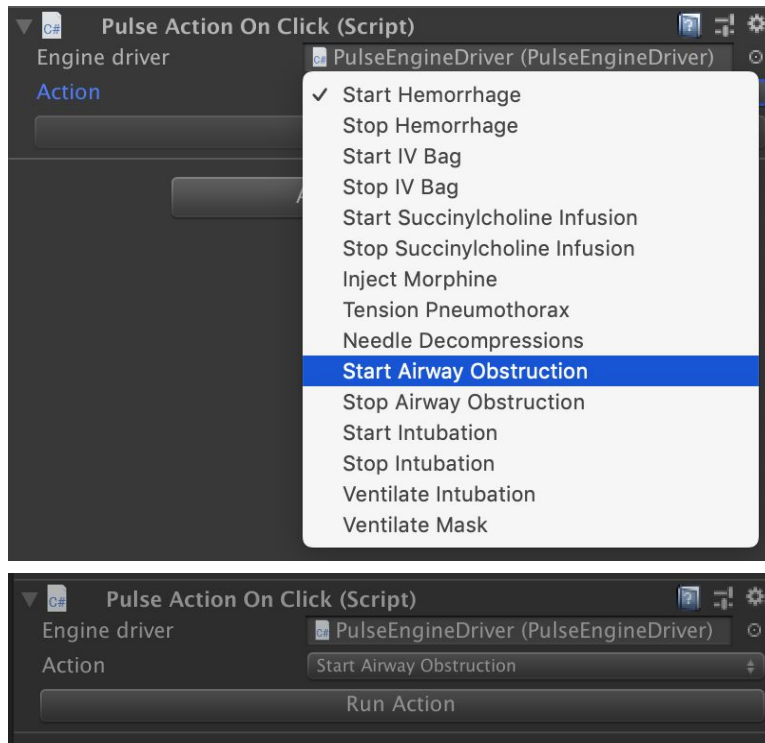
### VitalsMonitor

This example scene showcases a full vitals monitor screen similar to the [Pulse Explorer](#), constructed with multiple PulseDataLineRenderer and PulseDataNumberRenderer. Its input data is generated by a PulseEngine through the PulseEngineDriver component.

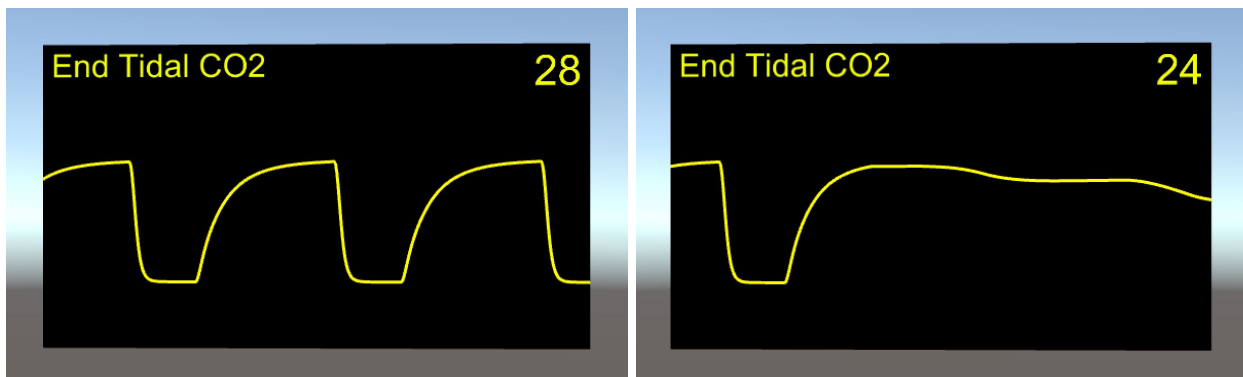


## CreateActionOnClick

This scene showcases a simple canvas in world space with a single `PulseDataLineRenderer` and `PulseDataNumberRenderer`, using data exposed through a `PulseEngineDriver`. The `PulseActionOnClick` component is added into the scene to be able to apply an action from a list of actions by pressing the “Run Action” button when the simulation is running. These examples provide example uses of each exposed action and inputs that can be used to simulate patient injury and treatment.



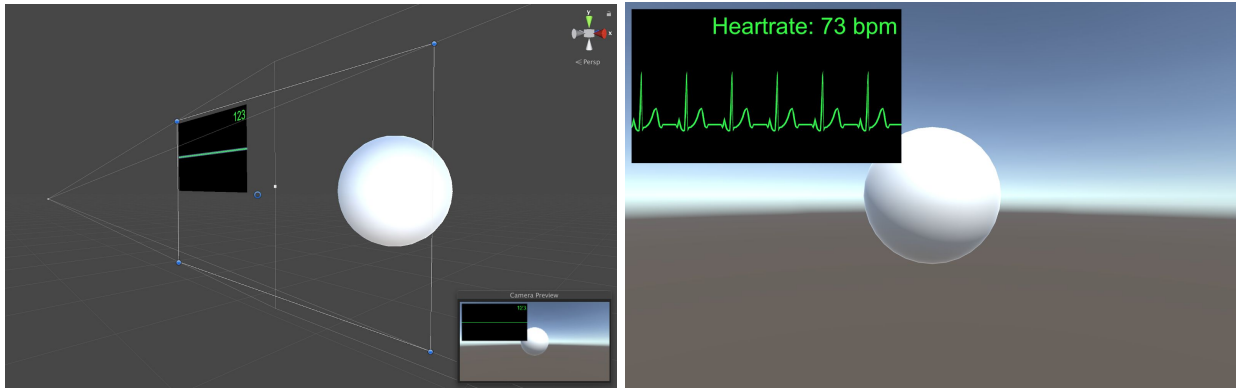
**Top: Selecting an action in the `PulseActionOnClick` editor;**  
**Bottom: `PulseActionOnClick` editor with enabled “Run Action” button when running.**



**Left: Capnogram and End-Tidal CO<sub>2</sub> of an healthy patient;**  
**Right: Same vital information after applying an airway obstruction.**

## ScreenSpace

This scene showcases a canvas in screen space (overlay) with a `PulseDataLineRenderer` and a `PulseDataNumberRenderer`, using data exposed through a `PulseCSVReader`.



**Left: scene view with canvas overlayed on the camera plane; Right: game view.**

## RandomNumber

This scene showcases a simple canvas in world space with a single `PulseDataLineRenderer` and `PulseDataNumberRenderer`, using data generated with a `PulseRandomValueGenerator`.

