Osaka University, Graduate School of Engineering, ERATO Asada Project

OpenSF

Open Skeleton Fitting: A generic framework for human skeleton tracking using depth cameras.

Norman Link <nlink@uni-koblenz.de>
Osaka, 29-Mar-12

CONTENTS

INTRODUCTION

The *OpenSF*-Framework (Open Skeleton Fitting) provides a framework to track human skeleton data using depth images. The depth images can be provided by various depth cameras like *Microsoft Kinect*, *Asus Xtion* or any other depth-map providing camera. A depth map is a dense representation of a scene's 3D point cloud. Using this data, common computer vision problems like different lighting conditions or segmentation can be avoided or simplified. Also, 3D information can be processed. Using a depth map of a scene containing a user, first the user is extracted. Then, on the user's depth silhouette, feature points are detected. Using these feature points, a skeleton is fitted inside the person using energy minimization. As of now (March 2012), no open source implementations of skeleton tracking algorithms are available. Current implementations like *NITE* or *MS Kinect SDK* are closed source and their details cannot be accessed. Thus, OpenSF can be used to develop independent skeleton tracking methods. OpenSF has been developed under the motivation to create an independent code base for further research in skeleton tracking. It has been developed as a research project for *Osaka University, Graduate School of Engineering, Japan* in the *ERATO Asada Project* of Prof. Minoru Asada and under supervision from Prof. Yukie Nagai. The framework is written in C++.

LICENSE

The source code of this project is not yet released under an open source license. The code may be used by Graduate School of Engineering, Osaka University and its respective students for research purposes and non-commercial applications. Releasing this project under an open source license is planned in the near future. For further information regarding the licensing of this project, please contact the author.

PREREQUISITES / DEPENDENCIES

OpenSF makes use of various external libraries. These are:

- Boost 1.44
  A utility library which contains a lot of helpful structures and algorithms. In OpenSF, Boost is used for quaternion mathematics and multithreading.
- OpenCV 2.3.1
  Open Computer Vision library for image processing.
- FLANN 1.7.1
  Fast Linear Approximate Nearest Neighbor library. In OpenSF, FLANN is used to compute nearest neighbor points in 3D.
- OpenNI 1.5.2.23
  Open Natural Interaction library. Provides a framework to access various kinds of depth cameras, including Microsoft Kinect and Asus Xtion Pro.
- GLFW 2.7.3
  A library that provides an abstraction layer from OpenGL and gives easy ways to handle 3D visualization and window management.
- OpenGL
  Open Graphics Library, a standard library for 3D visualization and rendering.

FRAMEWORK DETAILS

First, the general framework will be discussed as the base for further descriptions of implementation details.

The framework consists of different modules that are processed sequentially. Each module can generate output data of arbitrary type, which can then be used as input to the next module. A typical module has a base class and several implementation classes that derive from the base class. Every implementation class can define a different behavior. For example, the *Input* module can have the base class Input and the implementation classes *InputKinect* and *InputFile*. Therefore, every implementation class defines a different behavior, thus the interface of the base class stays the same (In this case, InputKinect would capture live input data from the Microsoft Kinect camera, while InputFile would read an input stream from a file stored on the hard drive). Instances of Modules are created by a *Factory* pattern, which identifies a specific module implementation based on a type identifier (`typeid`). Generation of the type identifier has to be done for every implementation class using the macro `MK_TYPE(classname)`. In the constructor, variables need to be initialized and the output and/or input slots have to be assigned. This has to be done by using the functions `addInput` and `addOutput`. Every module has to define at least the three virtual functions `iInit`, `isInit` and `iProcess`. The `iInit` function will be called once to initialize the module. Here, all the implementation-specific initialization and memory allocation should be done. The `isInit` function is called by the framework to check that the module has been initialized properly. This function should only return true, if everything was set up correctly and return false otherwise. When the framework starts running, the `iProcess` function will be called in every frame. Here, the module should process the data retrieved by the input pointers and store its results into the output pointers to provide the data to the next module. When the framework shuts down, all the modules will be deleted, causing their destructors to be called. Allocated memory should then be freed by the owning class.

Because this architecture involves a class hierarchy with *virtual functions*, it is very important to correctly call the corresponding superordinate function as suggested by the C++ standard. Otherwise, the class might not be initialized correctly because the calling hierarchy was disrupted. Also, correct memory allocation and deallocation is important to avoid memory leaks.

The current framework defines two executables *OpenSFApp* and *OpenSFVis*. OpenSFApp is a simple application to use the framework and retrieve data from it. OpenSFVis does the same, but also visualizes the retrieved data using in 2D and 3D using OpenGL.

COMPONENTS

The framework consists of the following components:

- *OpenSF*
  Contains the basic framework that all modules have to be based upon.
- *OpenSFLib*
  Contains the System class that manages and processes all the modules.
- *OpenSFInput*
  Contains the Input module to provide input data.
- *OpenSFSegmentation*
  Contains the Segmentation and Background modules that are responsible to segment a foreground user from the static background.
- *OpenSFFeatures*
  Contains the Features module that detects important feature points using the segmented foreground user.
- *OpenSFitting*
  Contains the Fitting module that uses the detected feature points as well as the user's point cloud to fit an arbitrary skeleton model inside the person by applying an energy minimization based inverse kinematics approach. The name OpenSFitting is missing an "F" in the middle on purpose, as the name would spell "Open Skeleton Fitting Fitting" otherwise.
- *OpenSFApp*
  A simple application that creates and initializes the pipeline and processes the data. No visual output is created.
- *OpenSFVis*
  Creates the same application as OpenSFApp, but the obtained results are visualized using 2D and 3D visualization.

MODULES

INPUT (OPENSFINPUT)

The Input module provides input data to the framework. It retrieves data using a depth camera like Microsoft Kinect. To increase processing speed of the framework, the module runs in a separate thread and starts retrieving the next frame immediately after finishing the current frame, while the other modules are still processing. In the base class, the camera's projection matrix is computed to map points in 3D to the image plane in 2D. You can retrieve the matrix by using `getProjMat`. Sequences can be recorded using `startCapture` and `stopCapture` functions. These will record a video stream as a sequence of numbered files to a directory, which can be read using the *InputNumbered* implementation.

INPUTKINECT

Retrieves input data using the Microsoft Kinect camera. The depth and the color stream can be aligned using `setRegisterDepth`. If available (currently only on Windows), the motor angle can be set using `setMotorAngle`. If several cameras are connected, the right one can be chosen using `setDeviceIndex`. To record image streams with Kinect, call `startRecording` and `stopRecording`, to write OpenNI-Files (.oni) to the hard drive.

INPUTNUMBERED

Retrieves input data from a sequence of numbered files from the hard drive. In combination with Kinect, this module should not be used, as the internal Kinect recording (.oni) provides a better way.

INPUTPLAYERONI

Retrieves input data from an OpenNI-File (.oni) stored on the hard drive. Set the filename with setFilename and the playback speed with `setPlaybackSpeed`. A speed value <= 0 plays the file as fast as possible. A value > 1 means playing fast-forward, a value between 0 and 1 means playing slow-motion. If the file should repeat from the beginning after the end is reached, call `setRepeat`. To pause the playback, call `setPauseMode`.

BACKGROUND (OPENSFSEGMENTATION)

The Background module is responsible for creating a background image out of the live image stream provided by the camera.

BACKGROUNDFIRST

Currently the only background implementation. It assumes the first image of the video stream to be the background images, which is supposed to contain only background objects. Only after the first frame, the test subjects should enter the scene.

SEGMENTATION (OPENSFSEGMENTATION)

The Segmentation module segments a user from the foreground and outputs an image that contains only the user, while the rest of the image is black.

SEGMENTATIONBACKGROUND

Currently the only segmentation implementation. It creates an instance of the Background module and uses the computed background image to segment the foreground user from the static background by applying subtraction and contour finding.

Parameters:

- `setThreshold` (Default: 0.15)
  Threshold for the background subtraction. The static background image is subtracted from the live depth image and then thresholded by this factor. The higher the threshold, the less noise the subtracted image will contain. It specifies how far a foreground pixel has to be from a background pixel. It is specified in meters.
- `setErodingSize` (Default: 3)
  After thresholding, the resulting image is eroded to remove some noise. This factor specifies the size of the eroding window.
- `setMedianBlurSize` (Default: 3)
  The image is then processed with a median filter to further remove small noisy pixels. This factor specifies the kernel size of the median filter.
- `setContoursFactor` (Default: 300)
  To extract the foreground user, a simple contour finding algorithm is applied. The largest found contour is considered the foreground user. To avoid tracking small noisy contours when no user has entered the scene, small contours are disregarded. The smaller this factor, the bigger a contour has to be to be considered a user.

FEATURES (OPENSFFEATURES)

The Features module computes important feature points using the segmented foreground user. The feature points are supposed to be those points on the user's body which are most distant from the center of mass. In the case of a human being, these points will be h*ead*, two h*ands* and two f*eet*. Together with the center of mass, which is considered to represent the user's t*orso*, there are ideally six feature points to be detected.

The detection is done using a graph algorithm on the user's body point cloud and computing the shortest distance from the center of mass to every point in the point cloud (*Dijkstra's Algorithm*). The result of the algorithm is called *Geodesic Distance Map*, because every pixel in the map represents the geodesic distance from this point to the center of mass. Then, by using the geodesic distance map, local maxima are computed to find the points in the map which are locally farthest from the user's center of mass.

After detection, the feature points are tracked over time to compute temporal information like velocity vectors and speeds. The tracking is done using a local assignment algorithm, which can later be improved by solving the global assignment problem (see the *Hungarian-Method*, also known as *Kuhn-Munkres-Algorithm*). In the current stage of development, the tracking is therefore not always optimal. The tracked feature points are also smoothed by using a Kalman-Filter to reduce the effects of noise.

Parameters:

- `setGeoMaxZDistThreshold` (Default: 0.1)
  To avoid wrong results, graph edges in the geodesic distance map are removed if their 3D distance is above this value. Higher values result in *islands*, black regions in the map for which no distances could be computed. Lower values result in wrong distances, when some body parts like arms are in front of the user.
- `setGeoNeighborPrecision` (Default: 8)
  Specifies, to how many neighboring pixels a graph node in the geodesic distance map can be connected to. Possible values are 4 and 8. A value of 4 means, every graph node can only be connected to its cross neighborhood. This is faster, but results in a lower quality of the distance map. A value of 8 means, a graph node can be connected to its full neighborhood.
- `setIsoPatchResizing` (Default: 160 x 120)
  To detect local maxima in the geodesic distance map, the map is undersampled to create *Geodesic Iso-*

Patches. These are regions of approximately the same distance from the center of mass. To reduce noise and to speed up the process, the distance map is resized using this parameter.

- `setIsoPatchWidth` (Default: 0.2)
  This parameter specifies the distance range within each geodesic Iso-Patch. It means that within each patch, only distances with ± X meters can occur. The higher this value, the fewer patches will be generated and the bigger a single patch can be. It also means that noise has a smaller effect, but values too high can cause body parts to be missing in certain situations. Smaller values result in a finer sampling of the geodesic distance map, but more local maxima thus feature points will be detected.
- `setTrSearchRadius` (Default: 0.3)
  To track feature points over time, a local assignment problem is solved. For every feature point in the previous frame, the algorithm searches for the nearest feature point in the current frame within the search radius. Higher values result in possible mismatches, smaller values result in a lower maximum speed. If the value is too small and the person moves a body part too fast, the tracking cannot correctly assign the matching feature point and will create a new tracking id instead. This can be avoided by solving the global assignment problem (see *Hungarian-Method*).
- `setTrFtLifespan` (Default: 10)
  If a tracked feature point is lost, its position is extrapolated for a limited time and the algorithm tries to reassign this feature point. If no reassignment could be done within the lifespan specified by this parameter, the extrapolated feature point is removed.
- `setTrFtPointTempTimespan` (Default: 20)
  For every tracked feature point, temporal information like velocity vectors and speeds is computed, as well as cumulative (summed) temporal information computed over a series of input data. Cumulative temporal information can improve the skeleton fitting afterwards. This parameter specifies, how many frames should be used to compute cumulative temporal information.
- `setTrFtKfMeasurementNoise` (Default: 1e-5)
  The position of a tracked feature point is tracked by a Kalman-Filter to reduce the effects of noise. This parameter specifies the measurement noise used for the filter. Higher values result in a stronger smoothing.
- `setTrFtKfProcessNoise` (Default: 1e-6)
  This parameter specifies the process noise used for the Kalman-Filter. Smaller values result in a stronger smoothing.

FITTING (OPENSFITTING)

After the feature points have been detected and tracked, a skeleton can be fitted inside the body. The skeleton is fitted using a combination of energy minimization and iterative inverse kinematics. The used IK method is called *CCD* (Cyclic Coordinate Descent), which adjusts the joint angles in an iterative way starting from an end-effector joint and traversing further to the root joint. To track a skeleton, every joint has to define an energy function. This function has to be minimal at the optimal joint position. For an end-effector joint, the optimal joint position is supposed to be situated at a feature position. For inner joints which are not attracted to feature points, either nearest neighbor search on the user’s body point cloud or on the geodesic path from the corresponding feature point to the center of mass is used as energy function.

To keep the framework as generic as possible, arbitrary skeleton hierarchies can be tracked. Therefore, the tracking behavior is defined within every skeleton implementation class independently. Every skeleton class defines a skeleton hierarchy with several joints, connected by bones with a specified length, also constrained in their possible angles. Constraining the possible joint angles greatly reduces the search space for the energy minimization, eliminating impossible poses.

Together with the joint hierarchy, a skeleton implementation class defines several functions which define the actual tracking behavior:

- Energy-Functions
  An energy function is used to define the optimal position for a certain joint. It has to be minimal at the supposed optimal joint position and has to be higher, the farther away the joint is from the optimal position. It has to be a steady function and should be designed in such a way, that it has only one strong minimum. In most cases, an energy function will be designed as a three-dimensional squared distance from the current joint position to an assumed 3D position on the user's body.
- Classificator-Functions
  A classificator function is called by the framework to classify a certain feature point to a joint. At first, the framework does not know, which feature point might correspond to which joint in the skeleton hierarchy. In order for the energy function to define a correct energy, it is necessary for every joint to know the corresponding feature point. Therefore, every end-effector joint has to define a classificator function, which gets passed a feature point and returns true, if the current feature point can be classified to this joint.
- Extrapolator-Functions
  When the user moves body parts in front of the body, the geodesic distance map sometimes gets corrupted, producing no relevant feature points. In this case, the extrapolator function gets called to produce alternate positions. The extrapolator function would generally use a nearest neighbor search on the body point cloud and choose the most appropriate nearest neighbor point for this joint. For example, if the user moves his hands in front of the body, the extrapolator function will choose the foremost nearest neighbor until the geodesic distance map again produces valid feature points. In case of the head joint, the extrapolator function chooses the topmost nearest neighbor.

The skeleton fitting algorithm is implemented in the *SkeletonFitting* class. It uses a combination of inverse kinematics and energy minimization. The idea is to compute the skeleton energy once, then change a joint parameter and compute the new skeleton energy, computing an energy gradient. The gradient then gives information about the direction in which to change the joint parameter to get to the energy minimum. Iterative application of this procedure is done until the change in energy is below a termination threshold.

The implemented energy minimization is a simple algorithm which converges slow, thus also being responsible for high computation times. To reduce this problem, a *Levenberg-Marquard* energy minimization using *CMINPACK* has also been implemented. However, experiments have shown that this energy minimization generates a different skeleton fitting behavior and is not necessarily more efficient. Other minimization algorithms can also be implemented in order to evaluate computation times and possibly increase efficiency.

Parameter:

- `setNNDepthStep` (Default: 3)
  To reduce computation time for computing nearest neighbors, this parameter specifies how many pixels from the input depth map should be skipped.
- `setNNDepthMaxLeafSize` (Default: 15)
  The FLANN library constructs a KD-Tree, which can be searched for nearest neighbors very efficiently. This parameter specifies how many points from the depth map a leaf can maximally contain.
- `setNNGeoStep` (Default: 1)
  To reduce computation time for computing nearest neighbors, this parameter specifies how many pixels from the input geodesic distance map should be skipped.
- `setNNGeoMaxLeafSize` (Default: 15)
  The FLANN library constructs a KD-Tree, which can be searched for nearest neighbors very efficiently. This parameter specifies how many points from the geodesic distance map a leaf can maximally contain.
- `setNNGeoCutoffFactor` (Default: 0.5)
  For computing nearest neighbors along a geodesic line, it is possible to cut off a percentage of the

path from the center of mass. A value of 0.5 means the path is cut off after 50 % from the center of mass. A value of 1.0 means, the geodesic path is no longer visible, a value of 0 results in the whole path to be processed. Modifying this parameter reduces some problems with energy functions that use nearest neighbors on a geodesic line.

- `setFitCCDMaxIter` (Default : 1)
  The inverse kinematics method *CCD* uses an iterative algorithm to minimize the skeleton energy. Higher values increase the number of maximum iterations, increasing the tracking quality but also resulting in a higher computation time.
- `setFitCCDChangeThresh` (Default: 0.001)
  The inverse kinematics algorithm will also terminate, if the number of iterations is still smaller than the previous parameter but the change in energy is below this threshold.
- `setFitCCDMinimzeSize` (Default: true)
  By default, also the overall size of the skeleton is optimized to adjust to different body shapes. In some skeleton hierarchies, this can produce unexpected behavior and therefore can be deactivated.

Implemented skeleton classes:

SKELETONUPPERBODY

An upper body skeleton with 9 joints: head, neck, left shoulder, right shoulder, left elbow, right elbow, left hand, right hand, torso.

The head classificator function is based on the following parameters:

- `maxXZDistance` (Default: 0.1)
  The distance of the head feature point to the torso position on the XZ-Plane must not be higher as this parameter.
- `minLifetime` (Default: 10)
  Specifies, how long a feature point has at least to be tracked before being considered as a head feature point.
- `maxMeanSpead` (Default: 0.2)
  The head feature point must not move a lot. A feature point can only be classified to the head, if its mean speed is below this parameter.

The hand classificator function is based on the following parameters:

- `torsoHeightTolerance` (Default: 0.3)
  A hand feature point has to be situated above the torso position plus this tolerance value.
- `minLifetime` (Default: 5)
  Specifies how long a feature point has at least to be tracked before being considered as a hand feature point.
- `minSummedRelMeanSpeed` (Default: 0.05)
  A hand feature point has to have a minimum summed mean speed to be considered as a valid hand feature point. The mean summed relative speed is specified in meters per frame. A higher value means the hand has to move faster to be classified. Please note that this value also depends on the feature tracking algorithm and the parameter *TrSearchRadius*.

Additionally, hand classification only starts after the head feature point has successfully been classified. To distinguish between right and left hand, the two-dimensional line from torso to head is computed and the left hand feature point supposed to be situated on the right side of the image (vice versa for the right hand).

SKELETONLOWERBODY

A lower body skeleton with 8 joints: torso, pelvis, left hip, right hip, left knee, right knee, left foot, right foot.

The foot classificator function is based on the following parameters:

- `torsoHeightTolerance` (Default: 0.3)
  A foot feature point has to be situated beneath the torso position plus this tolerance value.
- `minLifetime` (Default: 10)
  Specifies how long a feature point has at least to be tracked before being considered as a foot feature point.

SKELETONFULLBODY

A full body skeleton composed of SkeletonUpperBody and SkeletonLowerBody. Energy-, Classificator- and Extrapolator-Functions stay the same.

SKELETONSIMPLE

A simple skeleton only consisting of 5 joints: head, neck, left hand, right hand, torso. The Energy-, Classificator- and Extrapolcator- Functions are the same as for SkeletonUpperBody.

SKELETONMANIPULATOR

A two-bone manipulator demonstrating the application of the skeleton tracking method to the correspondence problem in child development. Because the skeleton is not constrained by additional end-effectors, turn off the minimization of the skeleton size for tracking.

The end-effector classification is based on the following parameters:

- `torsoHeightTolerance` (Default: 0.2)
  A feature point to be classified has to be situated above the torso position plus this tolerance value.
- `minLifetime` (Default: 10)
  Specifies how long a feature point has at least to be tracked before being considered as a valid feature point.
- `minSummedRelMeanSpeed` (Default: 0.03)
  A feature point to be classified has to have a minimum summed mean speed to be considered as a valid feature point. The mean summed relative speed is specified in meters per frame.
- `minRelSpeed` (Default: 0.08)
  A feature point to be classified has to have a minimum actual speed to be considered as a valid feature point.