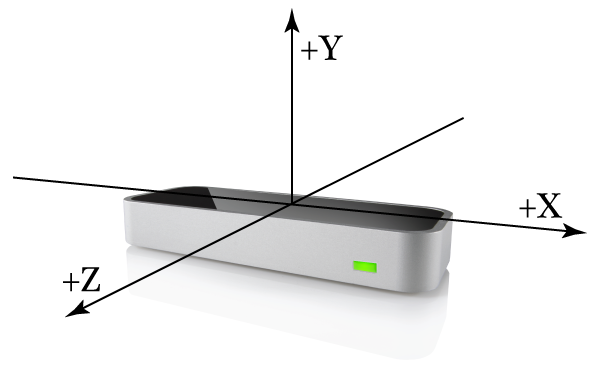
1) What is Leap Motion

The Leap detects and tracks hands, fingers and finger-like tools. The device operates in an intimate proximity with high precision and tracking frame rate.

The Leap software analyzes the objects observed in the device field of view. It recognizes hands, fingers, and tools, reporting discrete positions and motion. The Leap field of view is an inverted pyramid centered on the device. The effective range of the Leap extends from approximately 25 to 600 millimeters above the device (1 inch to 2 feet).

The Leap employs a right-handed Cartesian coordinate system. Values reported are in units of real-world millimeters. The origin is centered at the center of the Leap device. The x- and z-axes lie in the horizontal plane, with the x-axis running parallel to the long edge of the device. The y-axis is vertical, with positive values increasing upwards (in contrast to the downward orientation of most computer graphics coordinate systems). The z-axis has positive values increasing away from the computer screen.



As the Leap tracks hands, fingers, and tools in its field of view, it provides updates as a set, or frame, of data. Each frame contains lists of the basic tracking data, such as hands, fingers and tools describing the overall motion in the scene.

When it detects a hand, finger or tool, the Leap assigns it a unique ID designator. The ID remains the same as long as that entity remains visible within the device's field of view. If tracking is lost and regained, the Leap may assign a new ID (the software may not know that the hand or finger is the same as the one visible earlier).

2) GestIT

The library assigned to gesture recognition is named GestIT, and the language used for its implementation is F#.

The base idea of this library is to make possible any gesture recognition by developing a model based on Petri Net. Conceptually, as a Petri Net is based on two basic concepts of "places" and "transitions" (by which one or more tokens can travel), a gesture recognition can be shaped as a trip from a start point to an end point. The library supports the definition of gestures according to a specific metamodel, which enables the developers to define high level gestures while maintaining the possibility to decompose them in smaller parts, and to assign handlers to their subcomponents.

As written before, the meta-model allows the definition of a complex gesture starting from though *ground terms* and *composition operators*.

The ground terms represent features that can be tracked by developers for recognizing gestures. For instance, in a multi-touch application they are the events that allow tracking the finger positions (usually called touch start, move and end), while for full body gesture they are the events related to joint positions. Ground terms can be optionally associated to a predicate that has to be verified in order to receive the notification of a feature change.

The composition operators allow the connection of both ground terms or composed gestures in order to obtain a complex gesture definition. Their semantics can be summarized as follows:

* Iterative Operator: expresses the repetition of a gesture recognition an indefinite number of times.
* Sequence Operator: expresses that the connected sub-gestures (two or more) have to be performed in sequence, from left to right.
* Parallel Operator: expresses that the connected sub-gestures (two or more) must be recognized at the same time.
* Choice Operator: expresses that it is possible to select one among the connected components in order to recognize the whole gesture.
* Disabling Operator: expresses that a gesture stops the recognition of another one, typically used for stopping iteration loops.

3) LeapDriver

The purpose of the LeapDriver is to connect to Leap sensor, receive and collect its data in order to update the current state, and then to spread these information to the listeners who are interested in.

Leap makes it possible to listen to its updates by instantiation of two objects: a Controller, that is the mainly interface to the device, and a Listener, which makes a few methods available for connecting and receiving updates from the device. LeapDriver uses a *LeapListener* for this purpose. All the received data will be part of the history, which is a window of all frames received from 2/10s ago until current time. Each frame can be identified by its timestamp, which represent the time in microseconds.

Every frame of the Leap is a set of information about the world state at that current timestamp. In particular, the *Frame* object - as described before - is always the same instance with different data at every time, and that's why those informations are stored by LeapDriver into another *ClonableFrame* instance: the need for a history with many frames (Leap API doesn't give the possibility to instantiate *Frame* objects). This set of *ClonableFrame* objects maintains history from 2/10s ago up to current time.

Another important factor to keep in mind is that Leap can temporary lose visibility on its Hand and Pointable objects, which means that it could re-assign different IDs when tracking data is restored (please note that ID is the unique identifier of every current object on the scene, and it is unique until that object disappears to Leap's view - then the ID could be reassigned to another one, or the object could came back on scene with another ID). LeapDriver tries to fix this problem by maintaining in the history also "zombie" hands or pointables for a time of 2/10s: during this period, LeapDriver tries to match new objects with the zombie ones, to figure out if they could be the same. In this way, the unique identification of objects is safe, and this will especially help applications which are based on gesture recognitions. Please note that the information about zombie objects won't be notified anymore until they die - unless LeapDriver finds a match -, because we want the application to be notified only for *real* hands and pointables data.

The driver model can be listed in some parts as follows:

1. Truncate history: every time a frame is received, we need to clean history from too old information (especially zombie).
2. Prediction: after saving *Frame* object into a new *ClonableFrame* one, LeapDriver makes a simple prediction about what will be the new position of every object in the scene. This will help keeping alive the zombies.
3. Update: for every object in new frame, if it's already part of history, its information are updated with new ones (this means that predictions will be useless in this precise case). If there are not updated objects - because they're new on the scene - LeapDriver insert them into a particular list of not-updated elements.
4. Correct: the elements of that list are potential zombies coming back to life. So LeapDriver tries to match them - grounding on position data - with elements in history that aren't on scene anymore. If match is successful, old zombie item is replaced with the new one, except for the ID field that is maintained the same.
5. Extend: finally, for objects who aren't recognized as zombies, LeapDriver only has to update current state adding them as new elements on scene.

LeapDriver's work is also about to inform applications of different events coming from Leap sensor. When an event needs to be triggered, a *LeapEventArgs* is sent to applications listener: it will contain one kind of *LeapFeatureType*, plus the ID to which feature is referred.

There are different types of events that can be notified, so they've been classified into different *LeapFeatureType*:

* ActiveHand, ActiveFinger, ActiveTool: when one of these objects appears for the first time on scene, MoveHand, MoveFinger, MoveTool: when Leap reveals movements.
* NotActiveHand, NotActiveFinger, NotActiveTool: when one of these objects totally disappear from scene and especially from history.

4) How to use LeapDriver: LeapTrayApplication

LeapTrayApplication serves as an example of how to use LeapDriver. It's a simple application which permits a quick and intuitive interaction with the OS, and it easily integrates itself with mouse and keyboard.

First of all, it's necessary to create:

• a LeapSensor for receiving updates;

• a queue of *ClonableFrame* to keep a history and implement gesture recognition (please note that frames must be cloned);

• a net based on GestIT for gesture recognition.

As written before, this net is composed by ground terms and composition operators, which has to be created with appropriate library functions. It's important to highlight that a ground term can be associated with a predicate, which is the function that permits or inhibits the token transition to another ground term. For example, a ground term could be called "closeHand", and its predicate could be explained as "user is closing his palm into a fist".

To trigger a reaction after - or even during - gesture recognition, it is possible to set some handlers that will work on OS behaviour.