Welcome to Prepose!

Prepose is a domain specific language for building custom gesture recognizers. Eventually, we will evolve Prepose into a “Scripting Language for NUI.” We welcome feedback on how to improve Prepose.

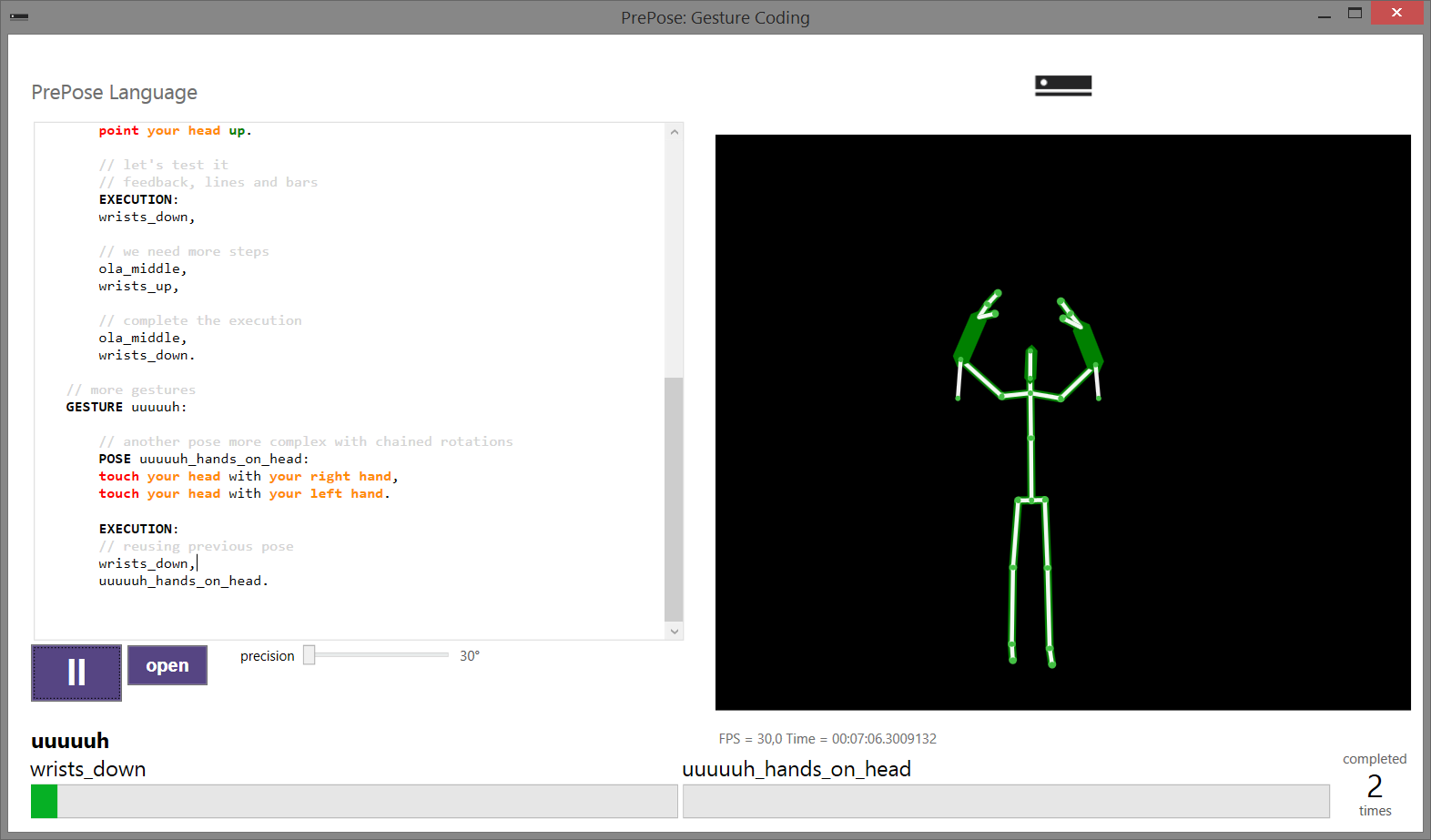
1. Obtaining Prepose

The code for Prepose is on Codebox at <http://codebox/prepose> . Run this to clone the repository:

**git clone** http://vstfcodebox:8080/tfs/EPSLION/\_git/prepose

Alternatively, use Visual Studio and connect to Team Foundation Server URL <http://vstfcodebox:8080/tfs/EPSLION> , project “Prepose”

Then follow these steps

1. Install the Kinect For Windows v2 SDK from  prepose\External\KinectSDK-v2.0-PublicPreview1407-Setup.exe
2. Open GestureRecognizer.sln in prepose  with **VS 2013**
3. Build and run “PreposeGestureRecognizer” by hitting “F5”
4. Running PreposeGestureRecognizer and writing your own Prepose code 

Poses in current gesture

Syntax-highlighted editor

User’s position (in green)

Target position (in white)

Current gesture

Load saved Prepose code

Precision tolerance

Start matching button

Above is a screen shot of the PreposeGestureRecognizer while running. On the left, a text editor window allows you to write code in the Prepose domain specific language with syntax highlighting and autocompletion. The highest level unit of Prepose is an “app.” Each app contains a set of “gestures,” which we will describe in more detail shortly.

When working with the PreposeGestureRecognizer, you write code for your gesture recognizer “app” in the text editor window. Then press the “start matching button” to parse your code and start interpreting the code. On the right, you will see a view of any Kinect skeletons detected by your Kinect v2 sensor. Before you start matching, the Kinect skeletons will be shown in green.

After you start matching, you will see the name of the current gesture at the bottom of the main window. You will also see a white skeleton that represents the “target” – that is, the place the tracked skeleton needs to be to match that part of the gesture. The thickness of skeleton limbs shows how close or how far away from the sensor the skeleton joint is. To match, a skeleton joint must be in the same location and the same depth as the target skeleton joint.

Each gesture consists of a sequence of “poses.” After matching starts, the bottom of the main window will show the names of each pose and a progress bar for each pose. The progress bar will fill up as the tracked skeleton comes close to matching a pose. After a pose is matched, Prepose will synthesize a new target skeleton using the Z3 constraint solver. After all poses are matched, the “recognized count” in the bottom right of the window increments. Then matching re-starts from the first pose in the gesture.

Here’s an annotated example to show Prepose syntax and concepts.

// Declare a Prepose program named “soccer”

APP soccer:

// We are building the “ola” or “wave” gesture

// An app can have multiple gestures

GESTURE ola:

// A gesture is composed of poses

// Poses specify a target skeleton position

POSE wrists\_down:

// use the “point” verb on body parts

// Down means orient the joint below the origin point at the hips

point your left wrist down,

point your right wrist down,

// pointing down is not natural, rotation needed

rotate your left wrist 30 degrees to your front,

rotate your right wrist 30 degrees to your front,

point your head up,

rotate your head 20 degrees to your front.

// We need more than one pose declared, so let’s create the next pose

// NOTE: poses can be declared in any order, not necessarily the order

// the user performs them to do the gesture.

// We will specify the order of poses in the EXECUTION: section below.

POSE ola\_middle:

// Prepose supports groups of body parts

// “your wrists” is the same as “your left wrist” and “your right wrist”

point your wrists to your front.

// We need more than one pose, so let’s create another one

POSE wrists\_up:

//put your hands above your head.

point your wrists up,

point your head up.

// Execution gives us a sequence of poses to match in sequence

// Commas separate each pose in the sequence

// We can re-use any pose declared in the app

// that includes poses not declared in this gesture (not shown)

EXECUTION:

wrists\_down,

// we need more steps

ola\_middle,

wrists\_up,

// complete the execution

ola\_middle,

wrists\_down.

NOTE: when you push the “start matching button,” PreposeGestureRecognizer will start running the **last gesture defined** in the text editor window, i.e. the one nearest the end of the text. This means if you want to try out a gesture that isn’t the last one in the text, you need to comment out or remove gestures following the last gesture.

Finally, you can load text files containing Prepose code into the text editor. To do this, press the button marked “open.” You can then navigate your file system to find text files. The repository includes several Prepose files here, including gestures from ballet, therapy, and tai chi:

GestureRecognizerKinectV2\Z3Experiments\Z3Experiments\Tests

2 What’s in the Prepose Solution

The solution file contains four projects

1. PreposeGestures – this is the core of Prepose. This contains the parser for the Prepose language, in the folder “Parser.” The Parser folder contains a grammar “PreposeGestures.g4” and files which are auto-generated by feeding PreposeGestures.g4 to the ANTLR grammar code generator. The “Tests” folder contains sample gestures and poses written in Prepose. For example, the “soccer.app” contains code for the “ola” or “wave” gesture described above.  
     
   The “Gestures” folder contains code that performs the runtime interpretation and matching of a Prepose gesture. This code is responsible for interacting with the constraint solver “Z3” to obtain a “target” Kinect skeleton pose given a gesture and a current skeleton pose.

The “Analysis” folder contains code that performs static analysis on Prepose gestures. Ambiguity.cs checks pairs of gestures to see if there exist user actions that could cause both gestures in the pair to match. Safety.cs checks a gesture to see if there exists a circumstance that could cause the gesture to ask the user to do something “unsafe,” where “unsafe” is defined by restrictions on the person’s movement.

1. PreposeGestureRecognizer – This is an interactive editor and debugger for PreposeGestures code. See the description above. For gesture recognition, MainWindow.xaml.cs registers a callback with the Kinect BodyFrameReader which obtains the Kinect body, converts it into the correct form, and passes the converted body to the TestBody method in PreposeGestures.
2. PreposeGesturesFrameReader – this is a wrapper around the PreposeGestures API that exposes Prepose through an interface similar to the existing VisualGestureBuilder interface. This code has a lot of rough edges. For example this code does not dispose of objects properly, and it throws NotImplementedException if you try to obtain a ContinuousGestureResult from a frame. This is intended as a proof of concept to show how the Prepose Gesture API can be wrapped to fit into another API. We are happy to move toward whatever API makes sense for developers.
3. PreposeGesturesFrameReaderConsoleExample – a simple console example that uses the PreposeGesturesFrameReader wrapper. This example was adapted from an example due to Ben Lower on how to use the VisualGestureBuilder API.

2. The PreposeGestures API Overview

To use the PreposeGestures API, the developer first creates a PreposeGestures “App” object. The constructor of the App takes in either a filename from disk containing Prepose code, or a text string that is a valid Prepose program. For example, this creates an App object from a string “preposeTextString”:

PreposeGestures.App myApp = PreposeGestures.App.ReadAppText(preposeTextString);

The API exposes a class called BodyMatcher that encapsulates the state and code for matching a gesture against a sequence of Kinect skeleton frames. The constructor takes as arguments a PreposeGestures App object and a “matchPrecision” argument that controls how close the user must be to the target position before a match is accepted. For example, this creates a new object “matcher” from the myApp above and an int “matchPrecision” variable.

BodyMatcher matcher = new BodyMatcher(myApp, (int)matchPrecision);

Finally, BodyMatcher exposes a TestBody method, which takes a matcher object, a Kinect V2 skeleton, and a Boolean “jumpToNextPose.” The “jumpToNextPose” should be false for performing matching, we explain in more detail below. The Kinect V2 skeleton is represented as a dictionary with type IReadOnlyDictionary<Microsoft.Kinect.JointType, Microsoft.Kinect.Joint>

which can be returned, for example, from a Kinect V2 BodyFrameReader.

The return value of TestBody is a list of GestureResult objects:

List<GestureResult> res = BodyMatching.TestBody(matcher, kinectJoints, jumpToNextPose);

The GestureResult object contains information about which poses have been matched in a gesture, how often the gesture has been completed, and how close the user is to matching the current pose. The implementation of TestBody calls the Z3 constraint solver.

The intention of “jumpToNextPose” is to support a not yet finished feature where the Prepose runtime can, given a valid Prepose program, give the developer a sequence of Kinect skeletons that match each pose. To check a user’s position against the target position, set “jumpToNextPose” to false. If the “jumpToNextPose” is true, then the TestBody method will return in the GestureResult a representation of where the user should move to match the current pose. This target skeleton can then be visualized or inspected programmatically.

3. The PreposeGesturesFrameReader API Overview

We have wrapped the PreposeGestures API with a “PreposeGesturesFrameReader” API that closely mirrors the interface to VIsualGestureBuilder gestures. With this API, the developer first creates a PreposeGesturesDatabase by parsing a file of Prepose code. Next, the developer creates a PreposeGesturesFrameSource and adds gestures to it from the PreposeGesturesDatabase. Finally to check for a completed gesture, the developer registers a callback with a PreposeGesturesFrameReader.

The example initialization of this API is as follows:

private void Initialize()

        {

            sensor = KinectSensor.GetDefault();

            pgd = new PreposeGesturesDatabase("soccer.app");

            pgfs = new PreposeGesturesFrameSource(KinectSensor.GetDefault(), 0);

            foreach (var g in pgd.AvailableGestures)

            {

                if (g.Name.Equals("ola"))

                {

                    gesture = g;

                    pgfs.AddGesture(gesture);

                }

            }

            pgr = pgfs.OpenReader();

            pgfs.GetIsEnabled(gesture);

            pgr.FrameArrived += pgr\_FrameArrived;

            sensor.Open();

        }

The pgr\_FrameArrived callback then obtains a PreposeGestureFrame from the callback arguments. From the PreposeGestureFrame, the callback can obtain a DiscreteGestureResult which has the same signature as the VisualGestureBuilder DiscreteGestureResult. This APi gives a developer who is familiar with VisualGestureBuilder a similar way to use Prepose gestures.

The main drawback to the PreposeGestureFrameBuilder is that it does not expose the richer information available in a PreposeGestures GestureResult. For example, it does not expose which pose the gesture is in, or how close an individual gesture is to completion. Of course these could be exposed through additional methods on the PreposeGestureFrame, but that would cause the API to diverge from the current VIsualGestureBuilder API.