Rigged hand model for the Blender Game Engine

B. Katalin Szabó
Reactor Monitoring and Simulator Laboratory
Hungarian Academy of Sciences Centre for Energy Research
Budapest, Hungary,
Faculty of Informatics, University of Debrecen
Debrecen, Hungary
szabo.katalin@energia.mta.hu

Abstract — **In a lot of virtual reality applications, it is necessary to display and animate the user's hand in the virtual world. In game engines, armatures are often used for modeling parts or the whole of the human body. The process of creating the armature and assigning parts of the mesh representing flesh and skin to the "bones" of the armature is called rigging. After performing rigging, moving the armature will result in the movement of the mesh, thus the virtual hand (or other object) is animated. The Blender Game Engine (BGE) is one of the few open-source game engines available. The paper gives a detailed description of the process of successfully building a complex rigged hand model for the BGE and gives guidance for driving this hand model with input data from the Leap Motion hand movement detector. The rigged hand model has been implemented, using the hand mesh from the LibHand library and an armature specifically built to match the hand representation of the Leap Motion. The model will be used for navigation and interaction in the virtual model of a power plant control room.**

Keywords — rigging; hand model; Blender Game Engine; Leap Motion

I. INTRODUCTION

In the world of virtual reality, the term "rigging" is used for the preparations to move (animate) virtual objects. Often these virtual objects are the whole or parts of the human body, especially the human hand. Depending on the requirements of use and the technical possibilities of the actual application (a game, a virtual tour of an existing or imaginary place, an instructional application etc.), the complexity of hand models varies greatly. A rudimentary 3D hand model may consist of five cuboids or cylinders representing the fingers. A complex hand model is more realistic in shape, with sophisticated coloring, it may even display small freckles on the skin of the hand.

These virtual reality applications are often realized in game engines, where the more complex animated hand models are usually constructed from armatures, which can be regarded as "skeletons" consisting of bones. These virtual bones and the bone structure they belong to may quite closely resemble the real bones of a real hand, or they may bear only a passing resemblance to them. During animation, the bones themselves are usually not displayed. Instead, "flesh" and "skin" (which are – usually complex - meshes in the game engine), which have been "glued" to them, are displayed. In this context, rigging is the process of creating the armature and the "gluing" (assigning parts of the mesh to particular "bones" of the armature). After the rigging has been performed, moving the armature will result in the movement of the mesh, thus the virtual hand (or other object) is animated.

The Blender Game Engine is a component of Blender (*www.blender.org*), a free and open-source 3D creation suite (cf. [1], [2]) used for making real-time interactive content. The version used by the author is 2.76b, under the Windows 10 operation system.

The Leap Motion hand tracking device (*www.leapmotion.com*) was introduced in 2013. There have been attempts to integrate it into the Blender Game Engine as well. Simpler hand models, like the one published by Magnus Benjes at *www.magben.de/?h1=3d* work fine, but it is not easy to build for the BGE a more sophisticated, armature-based rigged hand model with which it is possible to simulate (and animate with Leap Motion data) a hand using a complex mesh instead of objects of simple geometry like cuboids or cylinders. Those people who tried that path have usually reported failure and switching to another game engine, mostly to Unity. The probable reason for the failures is the peculiarity and complexity of the way armatures are handled in the BGE. However, as Blender is completely free and open-source, and most alternative tools are not, a solution in the BGE could come in handy for those wishing to use open-source, free tools. This paper gives a detailed description of the process of successfully building such a hand model and gives guidance for driving this hand model with input data from the Leap Motion.

II. LEAP MOTION'S REPRESENTATION OF THE HAND

The Leap Motion is a relatively inexpensive but reasonably accurate, optical-based hand detector device which is connected with a cable to the USB port of a computer. It uses infrared LEDs and cameras and gives quite detailed position, orientation, length and width information about the individual bones of the hand, and also some gesture detection (fist etc.). It provides application programming interfaces (APIs) for various programming languages. A feature list can be found at *blog.leapmotion.com/getting-started-leap-motion-sdk/*.

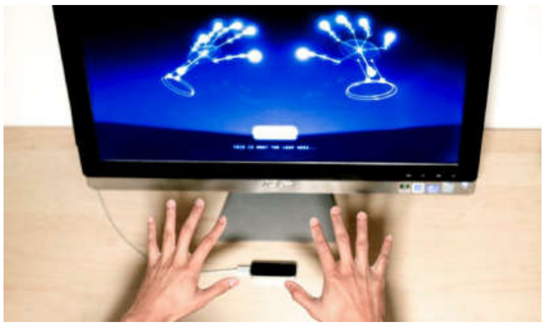
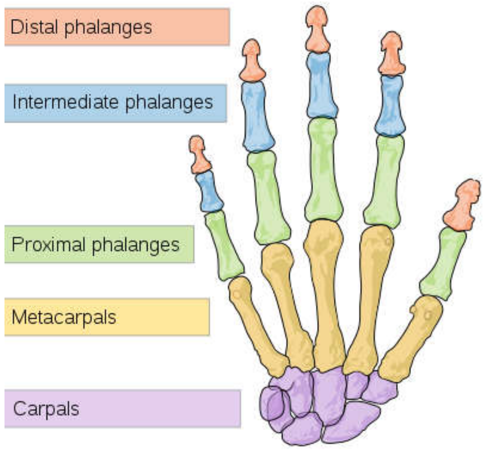


Fig. 1. The Leap Motion device [3]

The bones of the (right) human hand can be seen in Fig. 2 (with the palm facing the viewer).



Fig, 2, The bones of the human hand (*https://commons.wikimedia.org/wiki/File:Scheme_human_hand_bones-en.svg*)

Leap Motion's representation of the hand (including a part of the armbone) is in Fig. 3.

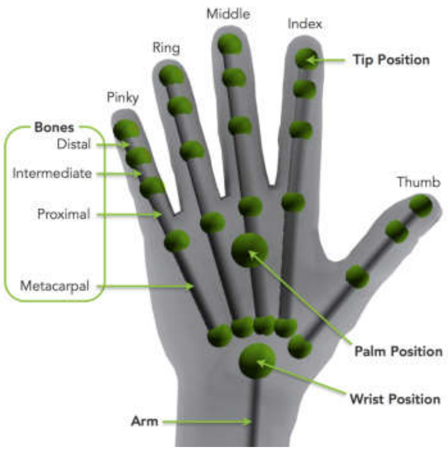


Fig. 3. Leap Motion's hand representation (adapted from *http://blog.leapmotion.com/getting-started-leap-motion-sdk/*)

III. A STEP-BY-STEP GUIDE TO THE RIGGING

This section describes the main steps of creating a usable rig for a relatively complex hand mesh in the Blender editor, to be used later in the BGE with the Leap Motion hand detector. Some familiarity with Blender's editor and the BGE is necessary for understanding the details. There are code samples (in Python which is used by the BGE) in the Appendix. The method can be adapted for hand meshes other than the one used here, and also for other hand movement detectors.

The hand mesh chosen is the mesh in the Blender file published at *www.libhand.org* (updates may be found at *github.com/libhand/libhand*). For the mesh, a new armature had to be constructed, to match the hand representation of Leap Motion.

The rig is for the right hand. You will later be able to derive the left hand from the right hand by mirroring the armature.

The basic idea behind this particular rigging solution is controlling the individual bones in the armature through *Copy Rotation* constraints which can be set in the Blender editor and will work in the BGE. For each bone, a cuboid of similar size is created. When the BGE application runs, each cuboid will get the orientation information for its corresponding bone from the Leap Motion device (through Leap Motion's Python API). Through *Copy Rotation* constraints, the orientation of each bone will be exactly the same as the orientation of the corresponding cuboid. However, due to the peculiarities of the BGE (some constraints are not completely or correctly implemented), these constraints will work properly only if the orientation of the armature never changes. It should always remain in an upright position (in a Z-up coordinate system), and only the orientation of its bones may change. (It should be noted that neither the armature itself, nor the bones can be seen when the virtual reality application runs, only the mesh “glued” to the bones of the armature is visible. Thus, regardless of the theoretical orientation of the armature, the bones will get the correct orientation. Consequently, the mesh will be correctly deformed and moved.) The position of the armature will change according to the position of the hand (as detected by Leap Motion).

The difficult part is to make a usable rig for the hand mesh so that the rest position of all of the armature's bones is the default upright position (otherwise the *Copy Rotation* constraints will not work correctly). The way to do it is first to pose the armature bones normally, so that they match the hand mesh that you use, then to do the rigging. After that, the armature bones should be posed to an upright position (with a script), By performing a trick with the modifier of the mesh, you can set this default upright pose as the new rest pose, without corrupting the rig. The resulting armature and the mesh rigged to it will look strange, but the mesh, controlled by the bones, which in turn are controlled by the orientation of the cuboids, will be animated correctly during the BGE run.

The cuboids used for the *Copy Rotation* constraints may have an additional function (provided that not only their orientation but also their position is updated according to the data taken from Leap Motion): if you need to handle collisions in your application, these simple cuboids, made invisible, may be used as a "shadow hand" which can collide with other objects, while the more complex hand mesh is left out of collision detection, thus conserving computer resources (this trick is often used in 3D games).

After this introduction, let us see the actual steps of the rigging.

A. *Make an armature which fits your mesh*

The way of creating armatures is described in Blender tutorials. First create the correct bone structure. During this first phase the exact bone lengths are not a priority, you should care about the correct topology only.

The next phase is setting the correct bone lengths. To facilitate this, you can make a run with the Leap Motion and print out the measured bone lengths, and you can set the bone lengths accordingly (manually or with a script).

For the mesh taken from *LibHand*, the armature in Fig. 4 has been constructed.

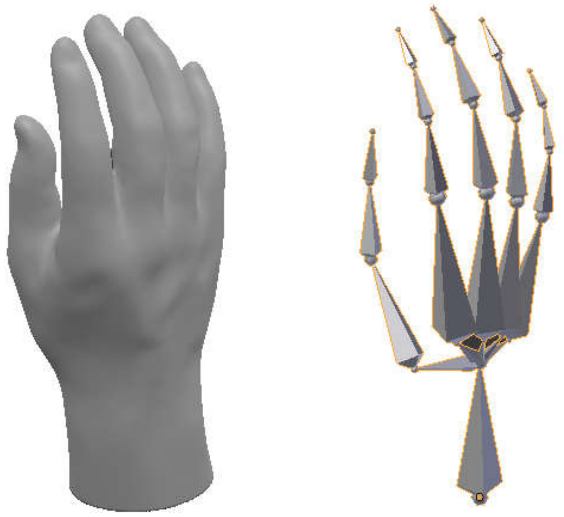


Fig. 4. Hand mesh and armature

It must be noted that, apart from the very first bone (the armbone), each bone has a "parent" bone to which it is connected. To ensure this continuous topology, certain "fictitious" bones had to be introduced, even though they have no corresponding bone in the hand model of Leap Motion. They connect the wrist position with the origins of the fingers. These fictitious bones differ in one thing from the "real" bones: they are set as non-deforming bones, so that the mesh will not be "glued" to them. (All other bones should be set as deforming bones.) The *Inherit Rotation* setting should be on for all bones. The lengths of the fictitious bones can be calculated from the end points of the connecting bones (which you can obtain from a Leap Motion run).

The third phase is to pose the bones so that they match the hand mesh as closely as possible. The posing can be aided by posing your real hand and taking note of the orientation of the bones as provided by the Leap Motion. You may use scripts for this but you will probably have to adjust some bones manually as well. This is usually an iterative process. Then perform "Apply Rotation" (for mesh and armature) in the Blender editor.

B. *Zero all bone rolls*

You may do this manually or with a script, in Edit mode (see sample script in the Appendix).

C. *Make the origin points the same for both armature and the hand mesh*

1) For each of these objects, set the origin to the object's lowest point center:
In Edit mode, move the 3D cursor to the desired origin point (to the point where the armature begins, the lowest round point in Fig. 5). Press Shift+S and choose the option Cursor to Selected.

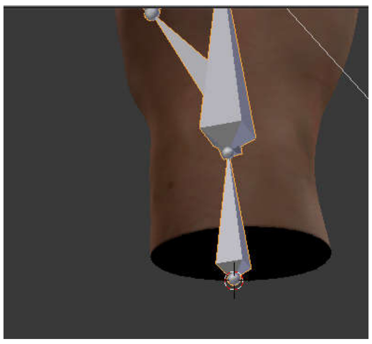


Fig. 5. Setting the origin point

In Object mode, select one of these objects and press ctrl/alt/shift C, then choose Set origin to 3D cursor.

Do the same for the other object.

2) In Object mode, position the hand mesh and the armature so that their origin points occupy the same place:
Make the XYZ coordinates of the two objects the same, by copying the coordinates of one into the other.

D. *Temporarily upscale mesh and armature*

This step is recommended because upscaled objects seem to rig better (at least when parenting is used with automatic weights). Make sure that no other object shares the space with the upscaled objects. Do the scaling (e.g. by a factor of 100) in Object mode, then apply scale.

E. *Perform automatic assignment of mesh points to the bones*

Parent the mesh to the armature: set Object Mode, select the mesh, shift-select the armature, press ctrl/p and select the "Armature Deform", "With Automatic Weights" option.

F. *Downscale the rigged armature*

In Object mode, scale the armature (not the mesh) back to its original size and apply scale.

G. *Align all bones to vertical*

This can be done with a script. First create an auxiliary armature with one bone being in a vertical position (that is the default). Then align each bone in the armature of the hand to this new bone (you can find a sample script in the Appendix).

H. *Make this pose the new rest pose*

This subsection is based on *nixart.wordpress.com/2013/03/28/modifying-the-rest-pose-in-blender/*.

In Object Mode, select your deformed hand mesh object.

In the object's Object Modifiers stack (denoted by the spanner symbol), copy the Armature Modifier by pressing the Copy button.

Apply the first Armature Modifier (the top one), but keep the bottom one. The latter will replace the old Armature Modifier and will allow to pose your object with respect to your new rest pose. At this point, the object will still be deformed twice. That is because we need to apply the current pose as the new rest pose.

Select your armature and set Pose Mode.

Select "Apply Pose as Rest Pose" in the Pose menu (which you can invoke by ctrl/a). This will clear the double deformation and put your object in your new rest pose.

I. *Create cuboids*

Create a cuboid object for each bone in the armature. All cuboids should have a vertical orientation. If you want to use these cuboids later for a "shadow hand", then make the lengths of the cuboids match the lengths of the armature bones.

J. *Create Copy Rotation constraints*

Set Pose mode. For each bone, create a Copy Rotation constraint to drive the bone in the armature with the rotation of the corresponding cuboid. The constraint subtype should be *World → Local with Parent*.

You can set these constraints manually, but it is easier to do it with a script (see Appendix).

IV. Interfacing with Leap Motion

Our rigged hand model is ready, now we want to drive it runtime in the BGE, with input from the Leap Motion device. Our task is to set the position of the armature and the rotation of the cuboids controlling the rotation of the bones.

A. *Interfacing the BGE with the Python API of Leap Motion*

The Python API of Leap Motion (Orion 3.2.0) uses Python 2.7, while Blender 2.76b uses Python 3.4. (Please note that different Blender versions may use different Python sub-versions.) Therefore, a Python wrapper has to be generated, this can be done with the SWIG interface generator, using the method described in *support.leapmotion.com/hc/en-us/articles/223784048* (the referenced page uses Python 3.3 but the method also works for Python 3.4).

B. *Driving the cuboids with Leap Motion data*

As already mentioned, an example with a simple hand model can be found at *www.magben.de/?h1=3d*. Its hand model consists of simple geometric shapes.

For a complex, armature-based hand model to work, we need detailed information from the hand movement detector. Fortunately, Leap Motion provides the information about how much the bones are rotated around their own longitudinal axis (this rotation is called the "roll" of the bone in armatures). To illustrate the importance of this, let us consider Fig. 6.

If we represent a finger with a simple cylinder, then the above mentioned rotation will not matter. However, if we make a fingernail on the cylinder (or "glue" a mesh, which includes a fingernail, to the cylinder), then this rotation will matter, as the fingernail will be visible.

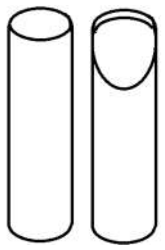


Fig. 6. Finger models without and with fingernail

The y basis vector is one of the three orthonormal basis vectors provided by the Leap Motion API (*developer-archive.leapmotion.com/documentation/python/api/Leap.Bone.html*). This vector is perpendicular to the longitudinal axis of the bone, it can be visualized as originating from the center of the fingernail and pointing outwards (in Fig. 6 the y basis vector would point from the fingernail directly towards the viewer). With the aid of this vector it is possible to set the bone roll for each finger correctly. This holds for all bones which have a representation in the Leap Motion hand model.

However, in our armature there are also fictitious bones, for which we get no basis vectors from Leap Motion.

C. *Calculating the orientation of the fictitious bones*

The orientation of such a bone (as a vector) can be calculated from its origin point and end point: the end of the armbone and the start of the metacarpal bone (these data are provided by Leap Motion). Thus we can set the orientation of the bone. The roll of the bone does not matter because the bone has been defined as a non-deforming bone, so no mesh ("fingernail") is "glued" to it.

V. Conclusion

The hand model which has been described above has been tested in the BGE and it reasonably accurately follows the user's hand movements, for the right and the left hand as well.

The use of such a hand model is especially perspectivic in immersive virtual environments, with the user wearing a headmounted display and not seeing his/her real environment. In such a situation, he/she cannot see the input devices, either, and if these input devices require touching (like a keyboard, a mouse, a game controller), this touch should be performed either "blindly", or a virtual representation of the input device should be displayed in the virtual environment, and this would usually require displaying a virtual model of the user's hand as well. Other possible applications would use only the user's hand as the "input device": by controlling the virtual hand with his/her real hand, the user could interact with objects in the virtual space directly, and hand gestures could be used for navigation and other actions as well (example: clenching the hand into a fist could mean that the user wants to quit the program). In these applications, it is also advisable to display the virtual hand, to provide feedback for the user.

Potential fields of application are numerous, e.g. visualization of virtual spaces like historical monuments [4], gamelike applications used in medical rehabilitation [5], instructional applications for teaching and providing practicing opportunities for staff at a facility. The very first application will belong to the latter category: the virtual hand will be used for navigation and interaction in the virtual model of a conventionally equipped nuclear power plant control room [6]. The navigation will be performed by hand gestures. The visual representation of the user's hand in the virtual space provides feedback for the user in the navigation, and shows the hand's position and orientation, relative to the physical devices in the control room, for the interaction with these devices. The interaction with the virtual models of physical devices (switches and pushbuttons) of the control room can be realized by colliding the virtual hand with these devices and sensing pushing (of buttons) and turning (of switches) from the dynamics of the hand movement. As haptic (tactile) feedback devices are still in experimental phase, the feedback about the collision could be visually provided or indicated by audio effects. Thus staff could exercise and even experiment in the virtual control room, without having to use the real control room or a physical replica of it.

Acknowledgment

The author expresses her thanks to Dr. Gábor Házi, head of the Reactor Monitoring and Simulator Laboratory of the Hungarian Academy of Sciences Centre for Energy Research, and to the leadership and the Scientific Committee of the Hungarian Academy of Sciences Centre for Energy Research, for making this work possible.

References

[1] Roland Hess: The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender, No Starch Press San Francisco, CA, USA, 2007

[2] John M. Blain: The Complete Guide to Blender Graphics: Computer Modeling and Animation, 4th Edition, A K Peters/CRC Press, September 2017

[3] Gennadiy Donchyts, Fedor Baart, Arthur van Dam, Bert Jagers: Benefits of the use of natural user interfaces in water simulation, Proceedings of the 7th International Congress on Environmental Modelling and Software (iEMSs), June 15-19, San Diego, California, USA, https://www.researchgate.net/publication/263655945_Benefits_of_the_use_of_natural_user_interfaces_in_water_simulations

[4] A. Gilányi, M. Bálint, R, Hajdu, S. Tarsoly, I. Erdős: A Visualization of the medieval church of Zelemér, in 6th IEEE International Conference on Cognitive Infocommunications, IEEE, 2015, pp. 449–453

[5] A. Gilányi, E. Hidasi: Virtual Reality Systems in the Rehabilitation of Parkinson's Disease, in 7th IEEE International Conference on Cognitive Infocommunications, IEEE, 2016, pp. 301–305

[6] G. Házi, J. Páles: Virtuális vezénylő a paksi teljesléptékű szimulátorhoz (Virtual control room for the full-scope simulator of Paks), Nukleon, December 2013, in Hungarian, http://nuklearis.hu/sites/default/files/nukleon/6_4_146_Hazi.pdf

Appendix: Python code samples

These code samples facilitate the creation of the rigged hand in the Blender editor. They should be executed in Blender's Python console.

Zeroing all bone rolls with script:

Set Edit mode, select all bones of the armature and run the following script:

```
for bone in bpy.context.selected_bones:
        bone.roll = 0
```

Aligning a bone's orientation to that of another bone:

The following code has been taken from *blenderartists.org/t/visual-transform-helper-functions-for-2-5/500965#post1804788*:

```
import bpy
from mathutils import Matrix, Vector
from math import acos

def get_pose_matrix_in_other_space(mat, pose_bone):
    """ Returns the transform matrix relative to pose_bone's current
      transform space.  In other words, presuming that mat is in
       armature space, slapping the returned matrix onto pose_bone
      should give it the armature-space transforms of mat.
      TODO: try to handle cases with axis-scaled parents better.
    """
    rest = pose_bone.bone.matrix_local.copy()
    rest_inv = rest.inverted()
    if pose_bone.parent:
      par_mat = pose_bone.parent.matrix.copy()
      par_inv = par_mat.inverted()
      par_rest = pose_bone.parent.bone.matrix_local.copy()
    else:
      par_mat = Matrix()
      par_inv = Matrix()
      par_rest = Matrix()
    # Get matrix in bone's current transform space
    smat = rest_inv * (par_rest * (par_inv * mat))
    # Compensate for non-local location
    #if not pose_bone.bone.use_local_location:
    #    loc = smat.to_translation() * (par_rest.inverted() *
rest).to_quaternion()
    #    smat.translation = loc
    return smat

def set_pose_rotation(pose_bone, mat):
      """ Sets the pose bone's rotation to the same rotation as the given matrix.
         Matrix should be given in bone's local space.
      """
      q = mat.to_quaternion()
      if pose_bone.rotation_mode == 'QUATERNION':
        pose_bone.rotation_quaternion = q
    elif pose_bone.rotation_mode == 'AXIS_ANGLE':
        pose_bone.rotation_axis_angle[0] = q.angle
        pose_bone.rotation_axis_angle[1] = q.axis[0]
        pose_bone.rotation_axis_angle[2] = q.axis[1]
        pose_bone.rotation_axis_angle[3] = q.axis[2]
    else:
        pose_bone.rotation_euler = q.to_euler(pose_bone.rotation_mode)

def match_pose_rotation(pose_bone, target_bone):
    """ Matches pose_bone's visual rotation to target_bone's visual rotation.
        This function assumes you are in pose mode on the relevant armature.
    """
    mat = get_pose_matrix_in_other_space(target_bone.matrix, pose_bone)
    set_pose_rotation(pose_bone, mat)
    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.ops.object.mode_set(mode='POSE')
```

You can copy the above code into the Python console. Then, assuming that you want to align a bone named *armbone* in the armature named *Armature.006* to the vertical bone named *Bone* in *Armature.004*, execute the following code in Pose mode:

```
target_bone = bpy.data.objects['Armature.004'].pose.bones['Bone']
pose_bone = bpy.data.objects['Armature.006'].pose.bones['armbone']
match_pose_rotation(pose_bone, target_bone)
```

Creating a Copy Rotation constraint:

Assuming that you want to set a Copy Rotation constraint for a bone named *armbone* in the armature named *Armature.006*, and you want to constrain the bone's rotation to the rotation of a cuboid named *armbone_cuboid*:

```
import bpy
obj = bpy.data.objects
objbone =
bpy.data.objects['Armature.006'].pose.bones['armbone'].constraints
objbone.new('COPY_ROTATION')
objbone['Copy Rotation'].target_space = 'WORLD'
objbone['Copy Rotation'].owner_space = 'LOCAL_WITH_PARENT'
objbone['Copy Rotation'].target = obj['armbone_cuboid']
```