A Real-Time Mesh Animation Framework Using Kinect

Yirui Wu[1], Tong Lu[1,⋆], and Jiqiang Song[2]

[1] National Key Lab for Novel Software Technology,
Nanjing University, Nanjing, 210046, China
[2] Intel Labs China, 8F, Raycom Infotech Park A, Haidian, Beijing, 100190, China
wuyirui1989@163.com, lutong@nju.edu.cn, jiqiang.song@intel.com

Abstract. This paper proposes a real-time mesh animation framework to animate any kind of virtual characters by using Kinect. A deformation module is first predefined by constructing propagation fields and accordingly the isolines in each field for a character. During animation, motions captured by Kinect are retargeted to movements of users' specified handle constraints, which induce the deformation module to produce natural animation results. Our interactive framework is user-friendly and featured by real-time response for user motions and intuitive manipulations. The experimental animation results and the user study on typical characters demonstrate the effectiveness of our framework.

Keywords: Character animations, Kinect-based motion recognition, Surface-based deformation, Real-time interaction.

1 Introduction

Computer animation is one of the most active areas in computer graphics and multi-media modeling, aiming at producing visually desirable animation results under user specified constraints [1,2]. It has been widely applied in many applications such as art modeling, movie making, game development and virtual reality.

Most existing 3D animation systems [3] employ simplified skeleton representations of characters. The position of a segment on a skeletal model is generally defined by animation variables, named as Avars. By changing values of Avars over time, an animation system performs skeleton-based deformations [4] to compute exact positions and orientations of each mesh frame by frame. However, skeleton-based animation potentially causes many artifacts since the influence of each bone on surface is often defined independently with the local geometry of a mesh. Moreover, rigging numerous controllers on a skeleton model for properly reacting with changed Avars can be quite annoying and inefficient. Instead, mesh-based animation systems [5,6] adopt surface-based deformation algorithms [7,8] to perform the animation due to their capability of preserving geometry details under users' intuitive controls. Unfortunately, most mesh-based animation systems suffer from high computation cost for interactive animation purpose.

⋆ Corresponding author.

B. Huet et al. (Eds.): PCM 2013, LNCS 8294, pp. 245–256, 2013.

This paper proposes a novel mesh-based 3D animation framework, which integrates a Kinect-based motion recognition module and a pre-modeled surface-based 3D deformation module, to perform animation of any virtual character through real-time interactions. The motion recognition module first captures animators' joint motions using Kinect and then transforms joint motions to movements of user-specified handles according to user-customized mapping rules. The deformation module finally produces natural and desirable animation results in real-time by instancing with handle constraints and considering handle movements as inputs.

The main contributions of this paper are:

- The proposed framework only requires a small number of controllers and enables real-time interactivity in most procedures, which results in an efficient and user-friendly animation system, especially for non-professional users.
- According to users' customized mapping rules, any kind of virtual characters can be properly animated to match users' expectation.
- The use of Kinect reduces the system cost, while improving robustness and expansibility. Therefore, the proposed framework can be easily applied to various design scenarios.

The rest of the paper is organized as follows. Section 2 reviews the related work. The proposed animation framework is described in detail in Section 3. Section 4 reports the experiments and discusses the results, and finally Section 5 concludes the paper.

2 Related Work

Animation Methods. There are two approaches in skeleton-based animation for generating Avars, keyframing [11] and motion capture [12]. The former requires animators to set Avars at pre-defined keyframes, and then all intermediate frames are automatically computed by interpolation of keyframes. Unfortunately, complex actions are generally difficult and time-consuming to produce by keyframing methods. The latter makes use of live actions by capturing animator's motions with numerous cameras and sensors, and then motions are applied to animate virtual character directly. Motion capture can produce accurate, fast and smooth movements, which are appropriate for design scenarios that require realistic behaviors and actions. However, such an animation system always requires professional manipulators and expensive hardware.

Former mesh-based animation systems [5,6] perform animation of high-quality surface characters from laser-scanned motions or videos. They generally transform the input motion description into a moving template model, i.e. a mesh or a point cloud. Then the animator specifies a set of markers on the character to map the motion. Finally, their systems automatically generate motion retargeting of characters incorporating with various kinds of surface-based deformation algorithms [7,8]. However, these systems cannot facilitate interactive animation due to their high computation cost and difficult manipulation.

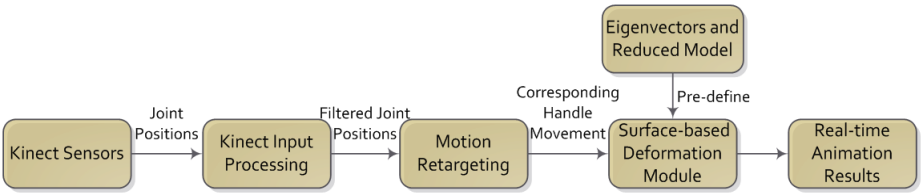

Fig. 1. The run-time work flow of our animation framework

Surface-Based Deformation Methods. In differential mesh editing [9,10], the deformation process is formulated as a global variational minimization problem to preserve local shape characteristics. Laplacian coordinates [13] are often adopted to describe these characteristics and should be properly transformed to match the deformed mesh. Otherwise, distortion like shearing and stretching will occur. After users specify handle constraints, the minimization problem is equivalent to solving the linear system in a least-squares sense:

$$\begin{gathered} ADG = b(DG), \qquad with \\ A = \begin{bmatrix} L \\ \Phi \end{bmatrix} \qquad and \qquad b(DG) = \begin{bmatrix} \sigma(DG) \\ C \end{bmatrix} \end{gathered} \tag{1}$$

where L is Laplacian matrix constructed from the original mesh before editing, D is the matrix representing transforming process, G is the transformation set, $\sigma(DG)$ is Laplacian coordinates which are nonlinearly dependent on the deformed vertex positions $X = DG$, and $\Phi G = C$ indicates the constraints of handle positions.

The handle-aware harmonic field method [9] computes transformations through solving a linear system defined by a discrete Laplacian operator with handle positions as its constraints. After sampling the transformations to obtain isoline sets as a reduced model, it updates reduced transformation domain in an iterative Laplacian editing framework [10,14], which achieves fast convergence, low per-iteration cost and stable deformation results. However, it requires a relatively long pre-computing time and high computation cost related to handle numbers if accepted for animation purpose.

3 Our Approach

Fig. 1 shows the run-time work flow of our proposed mesh-based animation framework. Our framework predefines a relatively independent deformation module based on the first few eigenvectors of Laplacian Matrix and the isoline-based reduced model before animation. During animation, we firstly use Microsoft Kinect as the source of Avars and properly filter the captured joint motions

to stabilize them. Next, joint motions are transformed to movements of user-specified handles on character according to user-customized mapping rules. Finally, deformation module is instanced by handle constraints and computes exact position of each vertex induced by handles movements. After rendering, we will obtain natural and desirable animation results.

3.1 Predefined Surface-Based Deformation Module

In this subsection, we propose a relatively independent module to pre-model deformation propagation before animation, which consists of two steps, *construct control variables* and *construct interpolation scheme*.

We first compute eigenvectors by eigen-decomposition of Laplacian matrix. Since the first eigenvector is a constant vector, we abandon it and sort the rest eigenvectors in ascending order of the corresponding eigenvalues. In fact, since a Laplacian matrix is mainly dominated by its low-frequency components [17], we can approximately describe a mesh by the first few eigenvectors of Laplacian Matrix. We thus select a fixed set of eigenvectors (vector number $n_e = 8$) as deformation propagation fields $\{e_0, e_1 ... e_7\}$, which describe the global geometric and topological information [15,16]. Then, we adopt an isoline-based reduced model [9] to construct isolines in each field (isoline number $n_{iso} = 10$), which serve as the generalized bones to imitate deformation propagation and will reduce run-time computation complexity. During animation, the transformation assigned to each isoline is regarded as *control variables* of each propagation field.

After constructing control variables, we adopt the same local interpolation scheme as Au. et al [9] to linearly compute transformation of each vertex from control variables in each propagation field. Furthermore, we define a high-level interpolation scheme to integrate transformation of different propagation fields in following three steps. First, we compute a weight factor for each propagation field by projecting one component of the original mesh coordinates to e_i:

$$\xi_{X,i} =< X, e_i >= \sum_{i=1}^{i=m} X \cdot e_i \tag{2}$$

where X represents the x, y, z coordinates of a mesh vertex and $\xi_{X,i}$ denotes the coefficients, which encode how propagation fields can reconstruct the original mesh geometry and more precisely the importance of propagation fields in representing the original shape. Second, we define the interpolation weight for each propagation field as follows:

$$w_i = \sqrt{\xi_{x,i}^2 + \xi_{y,i}^2 + \xi_{z,i}^2} \tag{3}$$

Finally, we normalize the weight set $\{w_i\}$ to make them sum up to 1 for the constancy of volume. The local interpolation scheme and high-level weight set $\{w_i\}$ make up a two-level *interpolation scheme* that provides parameterization for the original mesh space.

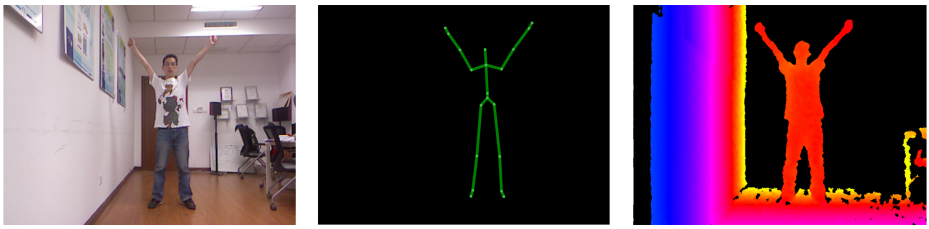

Fig. 2. From left to right: an animator captured by Kinect cameras, the extracted skeleton and the depth image which is transformed to HSL Space for a better view

We can represent the interpolation from the control variable to the vertex as $X = WR$, where R is a column vector constructed from isoline transformation set and W is a matrix constructed from $\{X\}$ and $\{e_i\}$, leading Eq. 1 to be updated with $D = W$ and $G = R$. Since W and L can be pre-computed before animation, we regard it as a relatively independent deformation module to describe the inherent constraints of deformation propagation.

3.2 Kinect Input Processing

Fig. 2 shows examples of a color frame, the extracted skeleton and the depth image captured by Kinect. However, the positions tracked by Kinect may be unstable and accordingly lead to artifacts. To avoid extreme situations and stabilize joint movements, we employ the following filter to process joint positions.

Joint Position Double Exponential Filter: This filter applies Holt-Winters double exponential smoothing [19] to joint positions captured by Kinect with the assumption that there would be a trend in the captured data:

$$\begin{cases} \tilde{p^1} = p^1 & t = 1 \\ b^1 = p^1 - p^0 & t = 1 \\ \tilde{p}^t = \beta p^t + (1-\beta)(p^{t-1} + b^{t-1}) & t > 1 \\ b^t = \gamma(\tilde{p}^t - \tilde{p}^{t-1}) + (1-\gamma)b^{t-1} & t > 1 \end{cases} \tag{4}$$

where p^t represents the original position of one specified joint at time t, b^t represents our estimate of the trend at time t, β is the smoothing factor and γ is the trend smoothing factor. Essentially, this filter adopts the previous historical locations to smooth the current locations. By adopting this filter, our system can remove jitters or noises and predict new joint positions to reduce lags, which leads to stable and desirable animation results.

Furthermore, we transform filtered joint positions to positions of application form based on the proportion between their sizes:

$$\tilde{p}^t = \frac{p^t * w^t}{u} - \frac{r}{2} \tag{5}$$

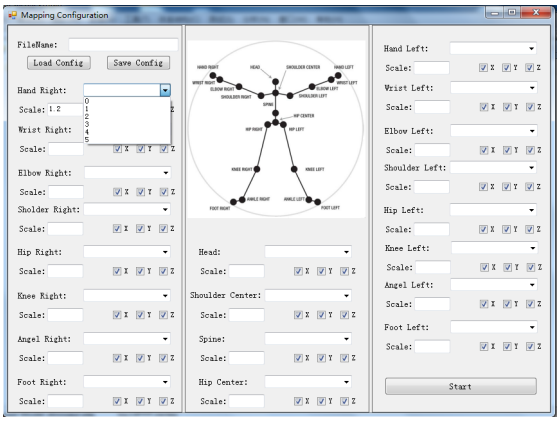

Fig. 3. The configuration panel that allows users to customize the mapping. Joints representing animator's skeleton are demonstrated in the middle to visually assist users.

where p^t represents x or y coordinate of filtered joint positions at time t, w^t and u respectively refer to the width or height of application form and captured frame, and r means the radius of handle point. Note that we accept the mean value of width and height as w^t for z coordinate.

3.3 Motion Retargeting to Handle Movement

In this subsection, we illuminate how to transform the filtered joint positions to the movements of user-specified handles based on user-customized mapping and animation parameters.

Users can intuitively manipulate handles on a mesh for animation. We thus generate a set of handle positions $\{h_0, h_1 ... h_{m-1}\}$ with handle number m. Considering that there is actually no exact correlations between parts of non-humanoid characters and animators' recognized joints, we allow users to customize the mapping between them in a configuration panel, which is shown in Fig. 3. In particular, the configuration incorporates the possible options related to the handle set $\{h_i\}$ and the joint set $\{j_k\}$. Kinect can recognize up to 20 joints demonstrated in the middle of the panel. Therefore, users' mapping can be formulated as a mapping matrix $M_{m\times 20}$, where the entry from ith row and kth column is set to one only if user defines the mapping between handle h_i and joint j_k, otherwise it's zero.

Furthermore, the configuration panel allows users to customize two parameters to scale joint motion $d_k^t = p_k^t - p_k^{t-1}$, the minus of joint positions between neighbor frames. Specifically, we can discard its x, y or z coordinate when the corresponding checkbox in the panel is unchecked. Therefore, we rewrite the joint movements as

$$d_k^t = f(p_k^t) - f(p_k^{t-1}) \tag{6}$$

where function f denotes the filter discussed in Section 3.2. We further compute the corresponding movement of user-assigned handle in the view of matrix:

$$H^t = M_{m\times 20} S^t_{20\times 3} \tag{7}$$

where S^t is constructed from the joint movements set $\{d^t_k\}$.

3.4 Real-Time Character Animation

Our character animation is based on the deformation module and the handle movements. When handles are specified by users, the deformation module will be instanced as Eq. 1. We solve the corresponding equation in a least square sense:

$$\begin{aligned} \tilde{R} &= (U^T U)^{-1}(W^T L^T \sigma(WR) + W^T \phi^T C) \\ X &= W\tilde{R} \end{aligned} \tag{8}$$

where $U = (W^T L^T \quad W^T \phi^T)^T$. Furthermore, we adopt an iterative Laplacian updating framework [9,10] to achieve the rotation invariant deformation by iteratively and alternatively updating R and X based on the current deformed surface. We also update laplacian coordinates $\sigma(WR^t)$ based on the rotation of triangles crossed by isolines in propagation fields. Therefore, Eq. 8 becomes:

$$\begin{aligned} R^{t+1} &= (U^T U)^{-1}(W^T L^T W_\delta R^t_\delta + W^T \phi^T C) \\ X^{t+1} &= WR^{t+1} \end{aligned} \tag{9}$$

where $\delta = W_\delta R_\delta$. Note that R_δ is a column vector constructed from the normal of the crossed triangles and W_δ is similar to former local interpolation scheme discussed in Section 3.1, i.e., linearly interpolating rotation for rest triangles. It is worth to recall that the constructed matrix W and L can be pre-computed before animation. So does the matrix W_δ. If we store constructed matrices $U^T U$ and $W^T L^T W_\delta$ at the first loading of any new model, and then reuse them for animation, our framework can allow instant animating of any pre-processed mesh.

In animation process, new handle positions are computed by $C^{t+1} = C + H^t$, involving corresponding joint motions to update the instanced deformation module. Since the definition of $\sigma(X)$ is separable in dimension of the vertices, the updating is performed separately for x, y, z coordinates of the deformed vertices. By applying isoline-reduced model, the original system matrix of size $n_{ver} \times n_{ver}$ is reduced to $U^T U$ of size $4n_{iso}n_e \times 4n_{iso}n_e$ after model projection from the original space to the isoline-based reduced space, speeding up the per-iteration updating significantly.

4 Results and Discussion

We animate a virtual character, Armadillo model, to present the effectiveness of our animation framework. Fig. 4 shows both the Kinect captured color frames

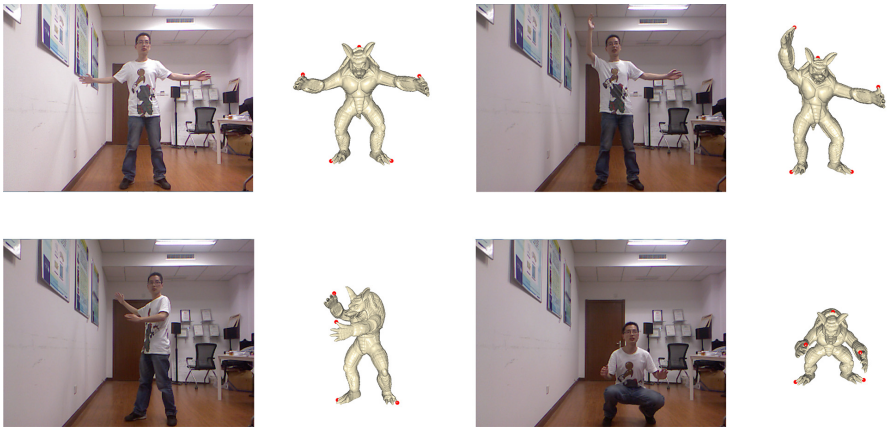

Fig. 4. Animation results of Armadillo model. Left: Kinect captured color frames. Right: Corresponding animation result.

and the corresponding animation frames created by our system. Note that high-frequency details are well preserved by Laplacian coordinates and deformation propagation on the surface is natural due to our well-constructed transformation matrix. In general, our system can capture 30 frames per second which ensures most captured human motions to be reflected in animation.

Table. 1 gives the detailed statistics on character and time performance of our animation system, measured on a 1.7GHz i5 core2 PC with 6GB of RAM. Thanks to the strategy of reusing transformation matrix as discussed in Section 3.4, we can reduce the time of pre-computing deformation module t_1 to a shorter time $\tilde{t_1}$, which is mainly about loading corresponding files and rendering character. We also achieve a rather low computation time $t_{2,3}$ for input processing and retargeting procedure. From Eq. 9, we can see per-iteration updating cost t_4 largely depends on the size of R_δ^t, which is related with the number n_{tri} of the isolines-crossed triangles and is often a relatively small number, resulting in a short enough per-iteration computing time to guarantee real-time response to 30Hz animator's motion input. Moreover, n_{tri} can be a fixed number for one specific model, which leads to a constant per-iteration time in spite of different number of handles and mesh vertices and thus relieves users' concern on causing not real-time response by defining a larger number of handles for animation. Therefore, our animation system achieves nearly instant system computation time $\tilde{t_1}$ and low per-iteration cost $t_{per} = t_{2,3} + t_4$, making the whole system efficient and user-friendly than the existing mesh-based animation systems.

4.1 User Study

We conducted user studies to evaluate the effectiveness of our framework. We observed ten novice animators (male and female, 10-30 years old), equally divided into two groups, Group One is familiar with 3D applications and Group Two

Table 1. Subscripts 1,2,3,4 respectively show time performance of each procedure of our animation system respected to different characters and handle numbers. $\tilde{t_1}(s)$ represents the computing time of the first procedure by reusing transformation matrices, P refers to the proportion of $\tilde{t_1}$ in the original t_1 and t_{per} represents per-iteration computing time

Model	n_{ver}	n_{han}	n_{tri}	$t_1(s)$	$\tilde{t_1}(s)$	$P(\%)$	$t_{2,3}(ms)$	$t_4(ms)$	$t_{per}(ms)$
Dinosaur	14000	6	841	2.917	**0.6132**	**21.02**	0.2353	**9.873**	**10.11**
Dinosaur	14000	10	841	2.915	**0.6411**	**21.99**	0.2185	**9.722**	**9.941**
Armadillo	60000	6	810	15.21	**3.327**	**21.87**	0.2540	**31.91**	**32.16**
Armadillo	60000	10	810	15.22	**3.307**	**21.73**	0.3021	**31.78**	**32.08**
Pegasus	32046	4	837	7.494	**1.704**	**22.74**	0.2785	**18.50**	**18.78**
Dog	18114	8	817	3.611	**0.8725**	**24.16**	0.2203	**11.38**	**11.60**
Human	15154	9	842	3.067	**0.7757**	**25.29**	0.2145	**10.62**	**10.83**

is not, to produce their animations with our system. We began with explaining defining handles and mapping rules with Human character. Next, they were free to play with our system for 3 minutes. After that, they were asked to animate four characters, Armadillo, Dinosaur, Pegasus and Dog. It's worthy noticing that Pegasus is a high-genus character which can not be well animated by the existing skeleton-based systems and the former two characters are humanoid characters which can be animated intuitively. The poses we set for characters are different. Armadillo acts a Kung Fu pose, Dinosaur leans its body, Pegasus lowers down its head, and Dog barks. We show two animation results completed by testers in Fig. 5. Particularly, for the Pegasus character, a user defines its head to correspond to his left hand, its tail to his right hand and the other two handles to two feet. In this way, he can easily achieve the animation in a natural and intuitive manner.

Testers further give a score s ranging from 1 (minimum) to 10 (maximum) to evaluate the accuracy, effectiveness and robustness comparing with the existing keyframing approach, which has been demonstrated to testers with detailed steps by the SketchUp program [20] cooperated with the Ruby plugin [21] and is scored as 5 in all aspects. Table 2 demonstrates the average completion time, total trial times with respect to different groups and testers' subjective evaluation. Although the number of testers involved is relatively small, their objective statistics and subjective impressions are important to understand the current limitation of our system from user's perspective.

The average score of our system 6.475, revealing that the Kinect-based approach do help to improve the productivity during the animation phase from user's perspective. Compared to the long learning time for a special Keyframing program, our system offers a more user-friendly way to perform animation easier and faster with a short completion time t_{avg}. Moreover, it is troublesome for users to control tiny changes of geometric details in Keyframing, while our system provides well-preserved and properly-transformed details automatically. Particularly, our system receives more approval rating, for animating the humanoid characters, Armadillo and Dinosaur, since users find it intuitive to define

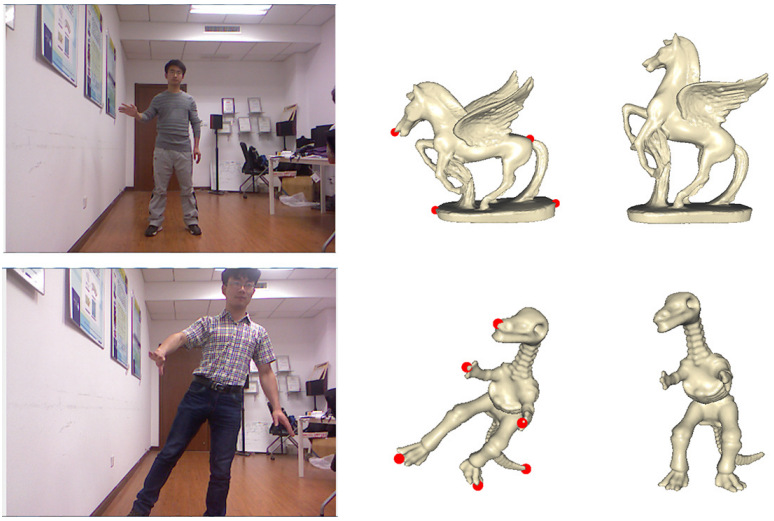

Fig. 5. *The first row* shows animation result of Pegasus character completed by a user from Group One; *The second row* shows result of Dinosaur character animated by a user from Group Two. We show original characters in the right of rows for comparison.

mapping and animation. This is reflected by the much shorter completion times of **1.606** and **1.732** min, and less trial times of **11** and **13** of these two characters. However, nine users try several times (total **35** times) for animating Pegasus, since most of them still apply the rules that map their heads to Pegasus's head, hands and feet to Pegasus's feet, which leads to undesirable animation results. Generally after four times of trials, we will give hints that he/she can control a specific part of a character with any joint and use a relatively small number of handles for animation, since most parts of characters can be animated by our system automatically. Testers can then complete animations following our given hints. Moreover, the tricks they learn from Pegasus can be quickly applied to Dog, which can be proved by a smaller trial times **16** and shorter completion time **2.522** min.

Table 2. Subscripts 1 and 2 show the objective performance in total trial times n and average completion time t (min) of Group One and Group Two. Pre-computing time of deformation module is excluded for fair comparison. We show subjective evaluations for animating compared with the Keyframing [20,21], which is scored as 5 in all aspects.

Model	n_1	n_2	n_{all}	t_1	t_2	t_{avg}	s_1	s_2	s_{avg}
Armadillo	5	6	11	1.476	1.736	1.606	6.5	7.5	7
Dinosaur	5	8	13	1.344	2.120	1.732	6.2	7.2	6.7
Pegasus	15	20	35	6.414	7.376	6.895	5.4	6.4	5.9
Dog	7	9	16	2.394	2.650	2.522	6.0	6.6	6.3

Our system is suitable for inexperienced users, since we find the trial times, completion time of Group One n_1, t_1 and Group Two n_2, t_2 are very close. We contribute this advantage to no further knowledge on 3D model and accordingly users can perform animation in an intuitive manner. For the robustness tested by multiple users, we generally obtain a lower score from Group One, which implies we need further improve our system to make it suitable for experienced users.

4.2 Implementation and Limitation

We use the skeleton and depth API provided by Kinect SDK to capture user motions. We adopt the cotangent weighting scheme to define the discrete Laplace operator, since it gives a better approximation to the normal of triangles. The solver of ARPACK is applied to accomplish the eigen-decomposition efficiently. We solve Eq. 9 by using Cholesky factorization of $U^T U$ for fast computing.

The use of only one Kinect sensor in the current solution could be an obstacle to the precision of our animation system. Particularly, occluded parts or clipped parts of the body can not be tracked accurately, which makes specific motion types hard to catch (e.g. a dancer spins around). Currently, an animator has to follow a few rules to successfully animate such a character.

5 Conclusion

This paper proposes a mesh-based framework for efficient 3D animation. It transforms Kinect captured motions to handle movements, and produces desirable animation results by processing handle movements in a surface-based deformation module. Our framework enables users to customize mapping rules between joints and handles to enable various animation styles. We believe our framework has a large potential for various design scenarios to reduce deployment cost and manipulation complexity, such as family movie-making systems and 3D games. Our further work includes motion recognition by using multiple Kinects.

Acknowledgments. The work described in this paper was supported by the Natural Science Foundation of China under Grant No. 61272218, the 973 Program of China under Grant No. 2010CB327903, and the Program for New Century Excellent Talents under NCET-11-0232.

References

1. Rick, P.: Computer animation - algorithms and techniques, 2nd edn. Morgan Kaufmann, New York (2007)
2. Robert, H., Ankit, G., Brian, C., Maneesh, A.: 3D puppetry: a kinect-based interface for 3D animation. In: UIST, pp. 423–434 (2012)
3. Andrea, S., Fabrizio, L., Gianluca, P., Felipe, D.R.: A kinect-based interface to animate virtual characters. Journal on Multimodal User Interfaces (2012)

4. Xiaohan, S., Kun, Z., Yiying, T., Mathieu, D., Hujun, B., Baining, G.: Mesh puppetry: cascading optimization of mesh deformation with inverse kinematics. ACM Trans. Graph. 26, 81–90 (2007)
5. de Edilson, A., Norimichi, U.: Representing and Manipulating Mesh-Based Character Animations. SIBGRAPI, 198–204 (2012)
6. de Edilson, A., Christian, T., Carsten, S., Hans-Peter, S.: Rapid Animation of Laser-scanned Humans. IEEE Vitual Reality, 223–226 (2007)
7. Mario, B., Olga, S.: On Linear Variational Surface Deformation Methods. IEEE Trans. Vis. Comput. Graph. 14(1), 213–230 (2008)
8. Mario, B., Leif, K.: Multiresolution Surface Representation Based on Displacement Volumes. Comput. Graph. Forum 22(3), 483–492 (2003)
9. Oscar, K.A., Hongbo, F., Chiew-Lan, T., Daniel, C.: Handle-aware isolines for scalable shape editing. ACM Trans. Graph. 26(3), 83 (2007)
10. Oscar, K.A., Chiew-Lan, T., Ligang, L., Hongbo, F.: Dual Laplacian Editing for Meshes. IEEE Trans. Vis. Comput. Graph. 12(3), 386–395 (2006)
11. Burtnyk, N., Wein, M.: Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation. Commun. ACM 19(10), 564–569 (1976)
12. Menache, A.: Understanding motion capture for computer animation and video games. Morgan Kaufmann, New York (2000)
13. Olga, S., Daniel, C., Yaron, L., Marc, A., Christian, R.: Hans-Peter Seidel: Laplacian Surface Editing. In: Symposium on Geometry Processing, pp. 175–184 (2004)
14. Yaron, L., Olga, S., Marc, A., Daniel, C., David, L., Christian, R., Hans-Peter, S.: Laplacian Framework for Interactive Mesh Editing. International Journal of Shape Modeling 11(1), 43–62 (2005)
15. Raif, M.R.: Laplace-Beltrami eigenfunctions for deformation invariant shape representation. In: Symposium on Geometry Processing, pp. 225–233 (2007)
16. Bruno, L.: Laplace-Beltrami Eigenfunctions Towards an Algorithm That "Understands" Geometry. SMI 13 (2006)
17. Zachi, K., Craig, G.: Spectral compression of mesh geometry. SIGGRAPH, 279–286 (2000)
18. Yizhou, Y., Kun, Z., Dong, X., Xiaohan, S., Hujun, B., Baining, G., Heung-Yeung, S.: Mesh editing with poisson-based gradient field manipulation. ACM Trans. Graph. 23(3), 644–651 (2004)
19. Gardner, E.S.: Exponential smoothing: The state of the art. J. Forecast., 4: 1C28 (1985)
20. SketchUp, `http://www.sketchup.com/`
21. Ruby plugin, `http://regularpolygon.org/keyframe-animation/`