Level-of-Detail Triangle Strips for Deforming Meshes

Francisco Ramos[1], Miguel Chover[1], Jindra Parus[2], and Ivana Kolingerova[2]

[1] Universitat Jaume I, Castellon, Spain
{Francisco.Ramos,chover}@uji.es
[2] University of West Bohemia, Pilsen, Czech Republic
{jparus,kolinger}@kiv.zcu.cz

Abstract. Applications such as video games or movies often contain deforming meshes. The most-commonly used representation of these types of meshes consists in dense polygonal models. Such a large amount of geometry can be efficiently managed by applying level-of-detail techniques and specific solutions have been developed in this field. However, these solutions do not offer a high performance in real-time applications. We thus introduce a multiresolution scheme for deforming meshes. It enables us to obtain different approximations over all the frames of an animation. Moreover, we provide an efficient connectivity coding by means of triangle strips as well as a flexible framework adapted to the GPU pipeline. Our approach enables real-time performance and, at the same time, provides accurate approximations.

Keywords: Multiresolution, Level of Detail, GPU, triangle strips, deforming meshes.

1 Introduction

Nowadays, deforming surfaces are frequently used in fields such as games, movies and simulation applications. Due to their availability, simplicity and ease of use, these surfaces are usually represented by polygonal meshes.

A typical approach to represent these kind of meshes is to represent a different mesh connectivity for every frame of an animation. However, this would require a high storage cost and the time to process the animation sequence would be significantly higher than in the case of using a single mesh connectivity for all frames. Even so, these meshes often include far more geometry than is actually necessary for rendering purposes. Many methods for polygonal mesh simplification have been developed [1,2,3]. However, these methods are not applicable to highly deformed meshes. A single simplification sequence for all frames can also generate unexpected results in those meshes. Hence, multiresolution techniques for static meshes are not directly applicable to deforming meshes and so we need to adapt these techniques to this context.

Therefore, our goal consists in creating a multiresolution model for deforming meshes. We specifically design a solution for morphing meshes (see Fig. 1),

M. Bubak et al. (Eds.): ICCS 2008, Part II, LNCS 5102, pp. 86–95, 2008.

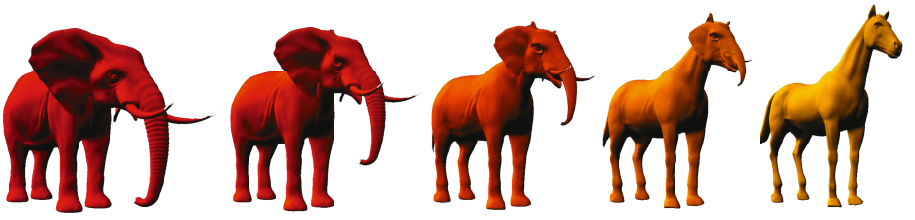


Fig. 1. A deforming mesh: Elephant to horse morph sequence

although it could be extended to any kind of deforming mesh. Our approach includes the following contributions:

- Implicit connectivity primitives: we benefit from using optimized rendering primitives, such as triangle strips. If compared to the triangle primitive, triangle strips lead us to an important reduction in the rendering and storage costs.
- A single mesh connectivity: for all the frames we employ the same connectivity information, that is, the same triangle strips. It generally requires less spatial and temporal cost than using a different mesh for every frame.
- Real-time performance: meshes are stored, processed and rendered entirely by the GPU. In this way, we obtain greater frame-per-second rates.
- Accurate approximations: we provide high quality approximations in every frame of an animation.

2 Related Work

2.1 Deforming Meshes: Morphing

A solution to approximate deforming meshes is to employ mesh morphing [4,5]. Morphing techniques aim at transforming a given source shape into a target shape, and they involve computations on the geometry as well as the connectivity of meshes.

In general, two meshes $M_0 = (T_0, V_0)$ and $M_1 = (T_1, V_1)$ are given, where T_0 and T_1 represent the connectivity (usually in triangles) and V_0 and V_1 the geometric positions of the vertices in R^3. The goal is to generate a family of meshes $M(t) = (T, V(t))$, $t \in [0, 1]$, so that the shape represented by the new connectivity T together with the geometries $V(0)$ and $V(1)$ is identical to the original shapes. The generation of this family of shapes is typically done in three subsequent steps: finding a correspondence between the meshes, generating a new and consistent mesh connectivity T together with two geometric positions $V(0)$, $V(1)$ for each vertex so that the shapes of the original meshes can be reproduced and finally, creating paths $V(t)$, $t \in [0, 1]$, for the vertices.

The traditional approach to generate T is to create a *supermesh* [5] of the meshes T_0 and T_1, which is usually more complex than the input meshes. After

the computation of one mesh connectivity T and two mesh geometries represented by vertex coordinates $V(0)$ and $V(1)$, we must create the paths. The most-used technique to create them is the linear interpolation [6], see Fig. 1. Given a transition parameter t the coordinates of an interpolated shape are computed by:

$$V(t) = (1 - t)V(0) + tV(1) \tag{1}$$

As commented before, connectivity information generated by morphing techniques usually gives rise to more dense and complex information than necessary for rendering purposes. In this context, we can make use of level-of-detail solutions to approximate such meshes and thus remove unnecessary geometry when required. We can also represent the connectivity of the mesh in triangle strips, which reduces in a factor of three the number of vertices to be processed [7].

2.2 Multiresolution

A wide range of papers about multiresolution or level of detail [8,9,10,11,12,13] that benefit from using hardware optimized rendering primitives have recently appeared. However, as they are built from a fixed and static mesh, they usually produce low quality approximations when applied to a mesh with extreme deformations.

Some methods also provide multiresolution models for deforming meshes [14,15,16], but they are based on the triangle primitive and their adaptation to the GPU pipeline is potentially difficult or does not exploit it maximally. Another important work introduced by Kircher et al. [17] is a triangle-based solution as well. This approach obtains accurate approximations over all levels of detail. However, temporal cost to update its simplification hierarchy is considerable, and GPU-adaptation is not straightforward.

Recent graphics hardware capabilities have led to great improvements in rendering. Mesh morphing techniques are also favored when they are employed directly in the GPU. With the current architecture of GPUs, it is possible to store the whole geometry in the memory of the GPU and to modify the vertex positions in real time to morph a *supermesh*. This would greatly increase performance. In order to obtain all the intermediate meshes, we can take advantage of the GPU pipeline to interpolate vertex positions by means of a vertex shader.

A combination of multiresolution techniques and GPU processing for deforming meshes can lead us to an approach that offers great improvements in rendering, providing, at the same time, high quality approximations.

3 Technical Background

Starting from two arbitrary polygonal meshes, $M_0 = (T_0, V_0)$ and $M_1 = (T_1, V_1)$, where V_0 and V_1 are sets of vertices and T_0 and T_1 are the connectivity to represent these meshes, our approach is built upon two algorithms: we first obtain a *supermesh* (Morphing Builder) and later we build the multiresolution scheme (Lod Builder). The general construction process is shown in Fig. 2.

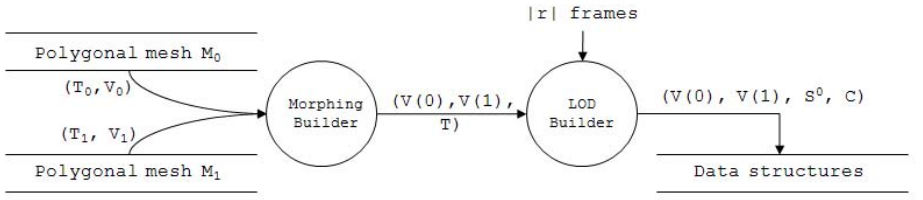


Fig. 2. General construction process data flow diagram

3.1 Generating Morphing Sequences

As commented before, linear interpolation is a well-known technique to create paths for vertices in morphing solutions. Vertex paths defined by this kind of technique are suitable to be implemented in recent GPUs offering considerable performance when generating intermediate meshes, $M(t)$. Thus, we first generate a family of meshes $M(t) = (T, V(t))$, $t \in [0, 1]$ by applying the method proposed by Parus [5]. As paths are linearly interpolated, we only need the geometries $V(0)$ and $V(1)$ and the connectivity information T, to reproduce the intermediate meshes $M(t)$ by applying the equation 1. The FaceToFace morphing sequence shown in Fig. 6 was generated by using this method [5].

3.2 Construction of the Multiresolution Scheme

A strip-based multiresolution scheme for polygonal models is preferred in this context as we obtain improvements both in rendering and in spatial cost. Thus, we perform an adaptation of the LodStrips multiresolution model [13] to deforming meshes. This work represents a mesh as a set of multiresolution strips. Let M the original polygonal surface and M^r its multiresolution representation: $M^r = (V, S)$, where V is the set of all the vertices and S the triangle strips used to represent any resolution or level of detail. S can also be expressed as a tuple (S_0, C), where S_0 consists of the set of triangle strips at the lowest level of detail and C is the set of operations required to extract the approximations. Every element in C contains the set of changes to be applied in the multiresolution strips so that they represent the required level of detail.

Thus, the process to construct the multiresolution scheme performs two fundamental tasks. On the one hand, it generates the triangle strips to represent the connectivity by means of these primitives, S, and, on the other hand, it generates the simplification sequence which allows us to recover the different levels of detail, C.

We generate the simplification sequence for each frame that we will consider in the animation. This task is performed by modifying the t factor in the *supermesh*. The number of frames to be taken into account is called $|r|$. The LOD builder subprocess first computes S, that is, it converts the *supermesh* into triangle strips. Later, it transforms the *supermesh* thus obtained into the subsequent intermediate meshes that we will use in the multiresolution scheme. We thus

store the sequence of modifications required into the triangle strips for each considered frame.

After the general construction process has finished, we obtain the sets $\{V(0), V(1), S^0, C\}$, where $V(0)$ and $V(1)$ comes from the *Morphing builder*, S^0 is the *supermesh* in triangle strips at the lowest level of detail, and C contains the sequences of simplification operations that enable us to change the resolution of the *supermesh* for each frame.

4 Real-Time Representation

Once construction has finished, we must build a level-of-detail representation with morphing during run-time. According to the requirements of the applications, it involves the extraction of a level of detail at a given frame. In Fig. 3, we show the main functional areas of the pipeline used in our approach.

The underlying method to extract approximations of the models is based on the *LodStrips* work [13]. Among other advantages already commented, this model offers a low temporal cost when extracting any level of detail for strip-based meshes. We take advantage of this feature to perform fast updatings when traversing a *supermesh* from frame to frame in any level of detail.

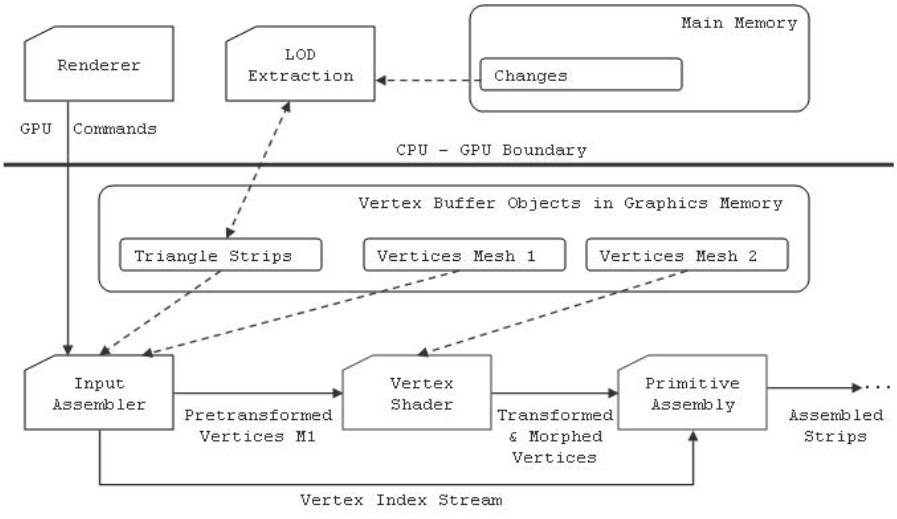


Fig. 3. Multiresolution morphing pipeline using the current technology

Thereby, according to the frame and level of detail required by applications, the level-of-detail extraction algorithm is responsible for recovering the appropriate approximation in the triangle strips by means of the previously computed simplification sequence. In Fig. 3, we show the general operation of this algorithm. It reads the simplification sequence of the current frame from the data

structure *Changes*, and it modifies the triangle strips located in the GPU so that they always have the geometry corresponding to the level of detail used at the current time. A more detailed algorithm is shown in Fig. 4.

After extraction, vertices must also be transformed according to the current frame in such a way that the deforming mesh is correctly rendered. When an application uses the GPU to compute the interpolation operations, the CPU can spend time improving its performance rather than continuously blending frames. Thus, by using the processing ability of the GPU, the CPU takes over the task of frame blending. Therefore, after extracting the required approximation, we directly compute the linear interpolations between $V(0)$ and $V(1)$ in the GPU by means of a vertex shader.

```
Function ExtractLODFromFrame (Frame,LOD)
  if Frame!=CurrentFrame then
    CurrentFrame=Frame;
    CurrentChanges=Changes[CurrentFrame];
    ExtractLevelOfDetail(LOD);
  else if LOD!=CurrentLOD then
    ExtractLevelOfDetail(LOD);
  end if
end Function
```

Fig. 4. Extraction algorithm

Regarding the GPU pipeline, the first stage is the *Input Assembler*. The purpose of this stage is to read primitive data, in our case triangle strips, from the user-filled buffers and assemble the data into primitives that will be used by the other pipeline stages. As shown in the pipeline-block diagram, once the *Input Assembler* stage reads data from memory and assembles the data into primitives, the data is output to the *Vertex Shader* stage. This stage processes vertices from the *Input Assembler*, performing per-vertex morphing operations. Vertex shaders always operate on a single input vertex and produce a single output vertex. Once every vertex has been transformed and morphed, the *Primitive Assembly* stage provides the assembled triangle strips to the next stage.

5 Results

In the previous section, we described the sets required to represent our approach: $V(0)$, $V(1)$, S^0 and C. According to the multiresolution morphing pipeline that we propose in Fig. 3, our sets are implemented as follows: $V(0)$, $V(1)$ and S^0 are located in the GPU, whereas C is stored in the CPU. In particular, $V(0)$ and $V(1)$ are stored as *vertex array buffers* and S^0 as an *element array buffer*, which offers better performance than creating as many buffers as there are triangle strips.

It is important to notice that we represent the geometry to be rendered by means of the data structures in the GPU, where the morphing process also takes

place. On the other hand, the simplification sequence for every frame is stored in the CPU. These data structures are efficiently managed in runtime in order to obtain different approximations of a model over all the frames of an animation.

Tests and experiments were carried out with a Dell Precision PWS760 Intel Xeon 3.6 Ghz with 512 Megabytes of RAM, the graphics card was an NVidia GeForce 7800 GTX 512. Implementation was performed in C++, OpenGL as the supporting graphics library and Cg as the vertex shader programming language.

The morphing models taken as a reference are shown in Fig. 6 and Fig. 7. The high quality of the approximations, some of which are reduced (in terms of number of vertices) by more than a 90%, can be seen.

5.1 Spatial Cost

Spatial costs from the FaceToFace and HorseToMan morphing models are shown in Table 1. For each model, we specify the number of vertices and triangle strips that compose them (strips generated by means of the STRIPE algorithm [7]), the number of approximations or levels of detail available, the number of frames generated and, finally, the spatial cost per frame in Kilobytes. It is divided into cost in the GPU (vertices and triangle strips) and cost in the CPU (simplification sequence). Finally, in the total column, we show the cost per frame, calculated as the total storage cost divided by the number of frames. As expected, the cost of storing the simplification sequence of every frame is the most important part of the spatial cost.

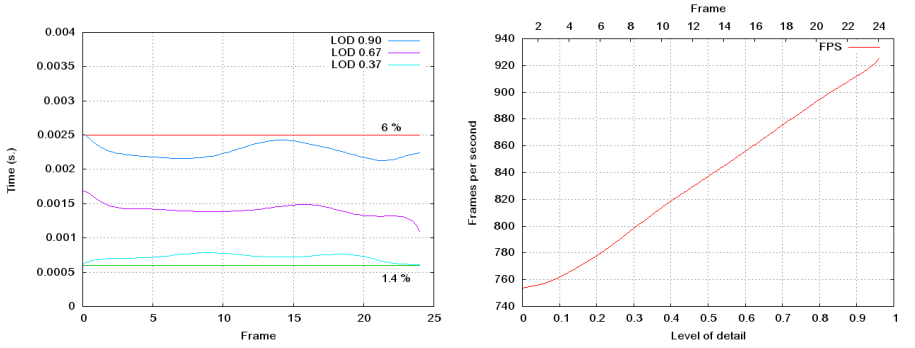
Table 1. Spatial cost

Model	#Verts	#Strips	#LODs	#Frames	Cost/Frame		
					GPU	CPU	Total
FaceToFace	10,520	620	9,467	25	14.2 KB.	472.1 KB.	486.3 KB.
HorseToMan	17,489	890	15,738	26	22.4 KB.	848.3 KB.	870.7 KB.

5.2 Temporal Cost

Results shown in this section were obtained under the conditions mentioned above. Levels of detail were in the interval [0, 1], zero being the highest LOD and one the lowest. Geometry was rendered by using the *glMultiDrawElements* OpenGL extension, which only sends the minimum amount of information that enables the GPU to correctly interpret data contained in its buffers. With *glMultiDrawElements* we only need one call per frame to render the whole geometry.

In Fig. 5a, we show the level-of-detail extracting cost per frame of the FaceToFace morphing sequence. The per-frame time to extract the required level-of-detail ranges between 6% and 1.4% of the frame time. If we consider the lowest level of detail as being the input mesh reduced by 90% (see LOD 0.9 in Fig. 5a), we obtain times around 6% of the frame time, which offers us better perfomance than other related works such as [17], which employs far more time in changing and applying the simplification sequence.



(a) Level-of-detail extraction cost per frame at a constant rate of 24 fps.

(b) Frame-per-second rates by performing one extraction every 24 frames.

Fig. 5. Temporal cost. Results obtained by using the FaceToFace morphing model.

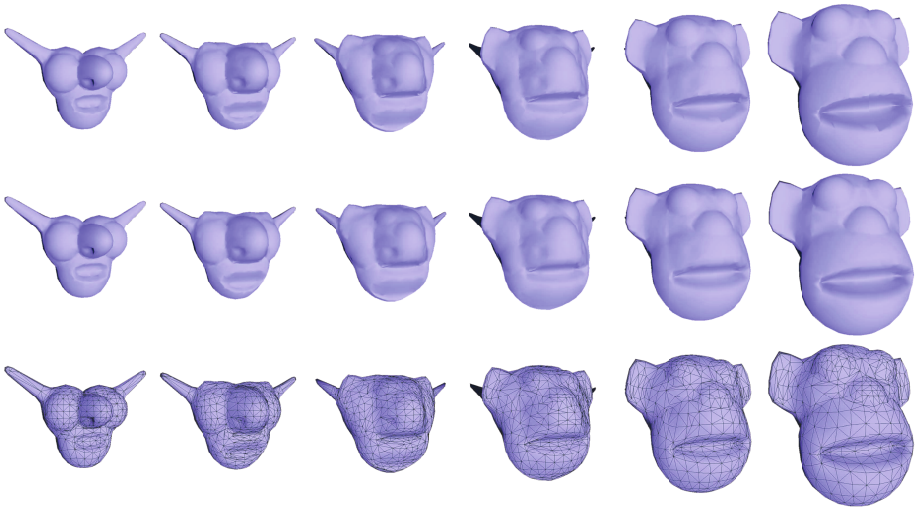


Fig. 6. Multiresolution morphing sequence for the FaceToFace model. Rows mean level of detail, 10,522 (original mesh), 3,000 and 720 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively.

We performed another test by extracting one approximation every 24 frames and, at the same time, we progressively changed the level of detail. This was carried out to simulate an animation which is switching its LOD as it is further from the viewer. In Fig. 5b, we show the results of this test. As expected, our approach is able to extract and render different approximations over all frames of an animation at considerable frame-per-second rates.

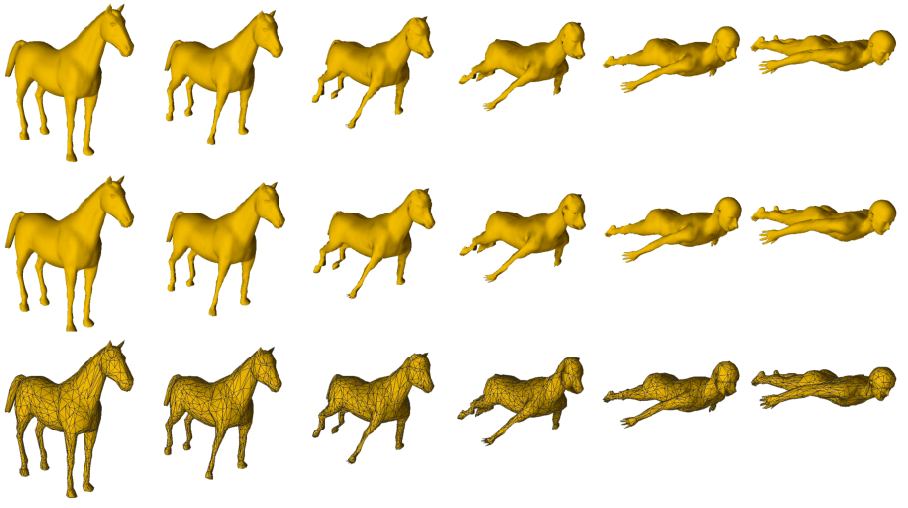


Fig. 7. Multiresolution morphing sequence for the HorseToMan model. Rows mean level of detail, 17,489 (original mesh), 5,000 and 1,000 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively.

6 Conclusions

We have introduced a multiresolution scheme suitable for deforming meshes such as those generated by means of morphing techniques. A solution for morphing sequences was specially designed, although it can be adapted to any kind of deformed mesh by storing the vertex positions of every frame within the animation. We also share the same topology storing the whole geometry in the GPU, thus saving bandwidth in the typical CPU-GPU bound bottleneck. Morphing is also computed in the GPU by exploiting its parallelism. We thus obtain real-time performance at high frame-per-second rates. At the same time, we offer high quality approximations in every frame of an animation.

Acknowledgments. This work has been supported by the Spanish Ministry of Science and Technology (Contiene Project: TIN2007-68066-C04-02) and Bancaja (Geometria Inteligente project: P1 1B2007-56).

References

1. DeHaemer, M., Zyda, J.: Simplification of objects rendered by polygonal approximations. Computer and Graphics 2(15), 175–184 (1991)
2. Cignoni, P., Montani, C., Scopigno, R.: A comparison of mesh simplification methods. Computer and Graphics 1(22), 37–54 (1998)

3. Luebke, D.P.: A developer's survey of polygonal simplification algorithms. IEEE Computer Graphics and Applications 3(24), 24–35 (2001)
4. Aaron, L., Dobkin, D., Sweldens, W., Schroder, P.: Multiresolution mesh morphing. In: SIGGRAPH, pp. 343–350 (1999)
5. Parus, J.: Morphing of meshes. Technical report, DCSE/TR-2005-02, University of West Bohemia (2005)
6. Kanai, T., Suzuki, H., Kimura, F.: Metamorphosis of arbitrary triangular meshes. IEEE Computer Graphics and Applications 20, 62–75 (2000)
7. Evans, F., Skiena, S., Varshney, A.: Optimizing triangle srips for fast rendering. In: IEEE Visualization, pp. 319–326 (1996)
8. El-Sana, J., Azanli, E., Varshney, A.: Skip strips: Maintaining triangle strips for view-dependent rendering. In: Visualization, pp. 131–137 (1999)
9. Velho, L., Figueredo, L.H., Gomes, J.: Hierachical generalized triangle strips. The Visual Computer 15(1), 21–35 (1999)
10. Stewart, J.: Tunneling for triangle strips in continuous level-of-detail meshes. In: Graphics Interface, pp. 91–100 (2001)
11. Shafae, M., Pajarola, R.: Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In: Pacific Graphics Conference, pp. 271–280 (2003)
12. Belmonte, O., Remolar, I., Ribelles, J., Chover, M., Fernandez, M.: Efficient use connectivity information between triangles in a mesh for real-time rendering. Computer Graphics and Geometric Modelling 20(8), 1263–1273 (2004)
13. Ramos, F., Chover, M.: Lodstrips: Level of detall strips. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3039, pp. 107–114. Springer, Heidelberg (2004)
14. Mohr, A., Gleicher, M.: Deformation sensitive decimation. In: Technical Report (2003)
15. Shamir, A., Pascucci, V.: Temporal and spatial levels of detail for dynamic meshes. In: Symposium on Virtual Reality Software and Technology, pp. 77–84 (2000)
16. Decoro, C., Rusinkiewicz, S.: Pose-independent simplification of articulated meshes. In: Symposium on Interactive 3D Graphics (2005)
17. Kircher, S., Garland, M.: Progressive multiresolution meshes for deforming surfaces. In: EUROGRAPHICS, pp. 191–200 (2005)