EUROGRAPHICS 2006 / E. Gröller and L. Szirmay-Kalos
(Guest Editors)

Volume 25 (*2006*), *Number 3*

Real-Time Weighted Pose-Space Deformation on the GPU

Taehyun Rhee[†1] J.P. Lewis[‡2] and Ulrich Neumann[§1]

[1]University of Southern California, U.S.A.
[2]Stanford University, U.S.A.

Abstract

WPSD (Weighted Pose Space Deformation) is an example based skinning method for articulated body animation. The per-vertex computation required in WPSD can be parallelized in a SIMD (Single Instruction Multiple Data) manner and implemented on a GPU. While such vertex-parallel computation is often done on the GPU vertex processors, further parallelism can potentially be obtained by using the fragment processors. In this paper, we develop a parallel deformation method using the GPU fragment processors. Joint weights for each vertex are automatically calculated from sample poses, thereby reducing manual effort and enhancing the quality of WPSD as well as SSD (Skeletal Subspace Deformation). We show sufficient speed-up of SSD, PSD (Pose Space Deformation) and WPSD to make them suitable for real-time applications.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture-Parallel processing, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling-Curve, surface, solid and object modeling, I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism-Animation.

1. Introduction

Skinning is an important part of realistic articulated body animation and is an important topic of computer graphics and animation. Generally, skinning can be categorized into algorithmic, physically-based, and example-based methods. Although widely used, simple algorithmic skinning schemes cannot capture the complexity and subtlety of real skin deformation, and revised approaches will be required to increase character animation realism. Physically-based skinning is based on the biomechanics of skin deformation arising from the motions of muscles and tendons. Although this approach can generate physically accurate simulations of each layer, it is not at present suitable for real time applications such as gaming due to the large computation required. Example-based methods capture some of the complexity of real skin deformation by interpolating scanned or sculpted examples of the desired skin shape in various poses. Although this requires gathering a sufficient number of samples and some pre-calculation, example-based methods can potentially be used in real-time applications due to their relatively simple real-time computation.

Weighted pose space deformation (WPSD) is an example based skinning method that generates high quality skinning with a limited number of sample poses [KM04]. Although it can generate an accurate skinning, it requires more computation than the original pose space deformation (PSD) [LCF00], since joint distances are computed independently for each vertex. As such, this method has not been suitable for real-time applications.

Furthermore, both WPSD and SSD require joint weights for each vertex, and accurate joint weights are required to achieve good results. However, the weights are usually manually generated by artists, which requires effort and great skill in the case of a complex skeletal system such as the human hand.

In this paper, we present a parallel WPSD algorithm (including automatic determination of joint weights) suitable for SIMD architectures such as current GPUs. The joint

† trhee@usc.edu
‡ zilla@computer.org
§ uneumann@graphics.usc.edu

weights for each vertex are automatically computed from the sample poses. This can enhance the skinning quality not only of SSD but also WPSD, since both methods require accurate joint weight values.

The deformation required in WPSD and SSD is independent for each vertex and this per-vertex computation can be parallelized in a SIMD architecture. The GPU is a general SIMD architecture having one-sided (unidirectional) communication to texture memory. We demonstrate our parallel WPSD method using GPU fragment processors. In our experiments, we can speed up SSD, PSD, as well as WPSD to around 20 times faster than on the CPU (from 1.2FPS to 25FPS speed-up of WPSD on a detailed model having 22836 triangles with 11574 vertices) using a modern graphics card, thus making WPSD a feasible real-time skinning solution for various applications including games, virtual reality, and other real-time simulations.

2. Related work

Many commercial software packages generate skin deformation arising from joint movement using a method known as (linear blend) skinning, Skeletal Subspace Deformation (SSD), enveloping, etc., based in part on work published by Thalmann et al. [MTLT88]. SSD is based on the weighted blending of affine transformations of each joint and used in many real-time applications due to its simple and fast computation. However, it also exhibits some well known artifacts such as skin that collapses around the joints at increasing bend angles, and a variety of solutions for these problems have been published [Web00, WP02, MTG03, KZ05].

Recently, example-based methods [LCF00, SRC01, ACP02, KJP02, KM04] have permitted more complex skinning effects such as muscle bulges and major wrinkles, while also addressing the artifacts of simple algorithmic schemes. In these methods, a number of provided (scanned or sculpted) samples of the desired skin shape are simply interpolated based on the creature's pose (and possibly additional abstract control "dimensions"). These example-based methods can also be considered as a non-parametric approach to skin deformation. In common with non-parametric sampling methods in texture synthesis (and more generally in statistical regression), the amount of memory for these methods grows with the number of training samples, but arbitrary distributions can be approximated.

Some of the most impressive example-based results to date are those of Kurihara and Miyata's hand model derived from medical images [KM04]. Since acquiring 3D medical images is relatively expensive, they developed weighted pose space deformation (WPSD) to generate proper skinning from a limited number of pose samples. They modify the distance between poses using the joint weights of each vertex to provide a more appropriate distance measure for skinning.

Although the joint weights for each vertex are important data for SSD and WPSD calculations, they have traditionally been manually generated by skilled artists. Least-squares based vertex weight estimation was shown in the skinning methods [WP02, MTG03]. James et al. describe mesh based skinning including estimation of bone parameters and vertex weights for each bone [JT05]. In their paper, the vertex weights of each joint are calculated by NNLS (non-negative least squares) and we derive a similar approach to calculate weights for SSD and WPSD.

In recent years, since the performance of GPUs has been improving more rapidly than that of CPUs, and GPUs have many processing units serving as a SIMD parallel architecture, many algorithms have been accelerated by GPU programming [LHK*04, PF05, GPG]. Deformation and skinning algorithms can also be enhanced by GPUs and several papers have profited from this [JP02, KJP02, BK05, JT05].

However, in previous research, since vertex information cannot be accessed in the fragment program, GPU-based vertex deformation is usually performed by vertex programs. In this paper, we develop a parallel WPSD method using the fragment processors to gain greater parallelism and performance.

Person-specific data modeling and its deformation is also an interesting topic in realistic articulated body simulation. Rhee et al. described human hand modeling from surface anatomy of the person [RNL06]. Anguelov et al. developed shape completion and animation of people, derived from the set of range scan data and example based deformation in pose and shape space [ASK*05].

Physically inspired skinning should be also recognized as another important area of articulated body animation. However, we entrust the review of the subject to the recent related papers [AHS03, CBC*05, PCLS05, SNF05].

3. Skin deformation

Example-based skinning problems can be described by the following general equation,

$$v(p_a) = S(v_0 + D(p_a)) \tag{1}$$

where p_a is an arbitrary pose, $v(p_a)$ is a vertex of a deformed target surface of the arbitrary pose, v_0 is an undeformed (rest pose) vertex, S is the SSD function, and $D(p_a)$ is a displacement as a function of the arbitrary pose.

In skeletal subspace deformation the displacement $D(p_a)$ is omitted and the target surface is calculated by SSD as a blend of affine transforms of v_0 [section 3.1]. Skinning methods related to PSD use the displacement of an arbitrary pose $D(p_a)$, calculated by interpolation in pose space [section 3.2].

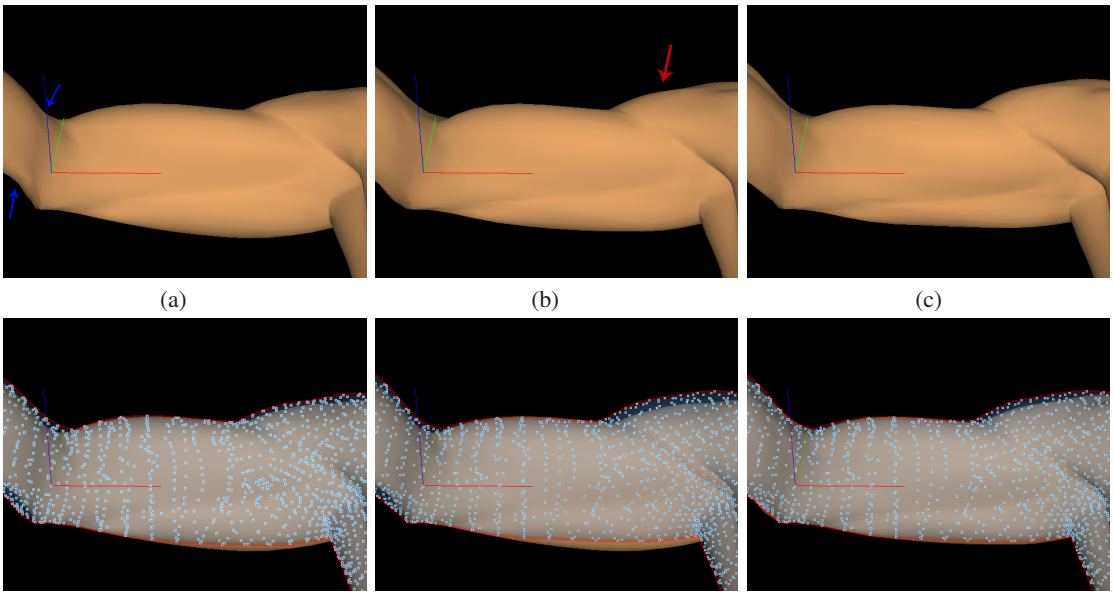

Figure 1: *Skinning result of each algorithm: (a) SSD, (b) PSD, (c) WPSD, (d) Difference between SSD and PSD (blue dotted area), (e) Difference between SSD and WPSD (blue dotted area), (f) Difference between PSD and WPSD (blue dotted area); areas around blue and red arrows represent unexpected results of SSD and PSD respectively.*

3.1. Skeletal subspace deformation (SSD)

SSD [MTLT88] is based on the weighted blending of an affine transformation of each joint by equation 2.

$$v_a = S(v_0) = (\sum_{j=1}^{n_{joint}} w_j T_j) v_0 \tag{2}$$

where n_{joint} is the number of joints, v_a is a vertex in an arbitrary pose p_a, v_0 is a vertex in the rest pose, and w_j is a joint weight that defines the contribution of joint j's transformations to the current vertex. The weight w_j can be assigned by the artist to control deformation and usually $\sum_{j=1}^{n_{joint}}(w_j) = 1.0$. This simple algorithm is used in many commercial graphics packages and real-time rendering applications but shows several limitations, because the deformation of this method is restricted to the subspace of the affine transformation of the joints [LCF00].

3.2. Pose space deformation (PSD)

If we have a sufficient set of examples to describe the movement of an articulated object, we can interpolate displacement in "pose space" [LCF00]. Each sample pose consists of sample skin geometry and the related joint skeleton, and a vector containing the joint angles represents the pose.

If we translate each skinning sample k to the rest coordinate frame using inverse SSD, the displacement between the sample vertex v_k and the rest pose vertex can be calculated:

$$d_k = (\sum_{j=1}^{n_{joint}} w_j T_j)^{-1} v_k - v_0 \tag{3}$$

where v_k is a vertex in sample pose p_k and d_k is the displacement of this vertex relative to v_0 in the sample pose p_k; the other variables are defined as in equation 2. Note that the inverse here is of the weighted sum of affine transforms.

After defining the displacement of each pose, the displacement at an arbitrary pose can be calculated by RBF (Radial Basis Function) [LCF00] or normalized radial basis function [KM04] interpolation of the example poses' displacements.

The displacement d_a of a vertex in an arbitrary pose p_a can be calculated as

$$d_a = D(p_a) = \sum_{k=1}^{n_{pose}} r_k(p_a) d_k \tag{4}$$

where n_{pose} is the number of sample poses, d_a is a displacement of the vertex in an arbitrary pose p_a, and the weight $r_k(p_a)$ defines the contribution of each sample pose.

Normalized Radial Basis Functions can smoothly interpolate pose space using:

$$f_t(p_a) = \sum_{k=1}^{n_{pose}} \lambda_{t,k} \phi_k(\gamma_k) \tag{5}$$

where $f_t(p_a)$ is the radial basis weight function for example t evaluated at an arbitrary pose p_a, n_{pose} is the number of

sample poses, $\lambda_{t,k}$ are real valued weights between pose t and k, ϕ_k are the radial basis functions, and γ_k is the distance between the pose p_k and the arbitrary pose p_a (defined as the Euclidian distance between the joint vectors of each pose).

The weight $r_k(p_a)$ is calculated using normalized RBFs and is used in equation 4 to calculate the displacement d_a of a vertex in an arbitrary pose p_a:

$$r_k(p_a) = \frac{f_k(p_a)}{\sum_{t=1}^{n_{pose}} f_t(p_a)} \tag{6}$$

Gaussian radial basis functions $\phi_k(\gamma_k) = exp(\frac{-\gamma_k^2}{2\sigma^2})$ are one possible choice of the basis and the constant σ can be specified experimentally [LCF00].

3.3. Weighted pose space deformation (WPSD)

WPSD is developed by Kurihara et al. [KM04] to deform their example-based human hand model derived from medical images. In the original PSD, the distance between two poses p_a and p_k having n_{joint} number of joints for each pose is defined as

$$\gamma_k(p_a, p_k) = \sqrt{\sum_{j=1}^{n_{joint}} (p_{a,j} - p_{k,j})^2} \tag{7}$$

In equation 7, since the γ_k is the difference of n_{joint}-dimensional joint vectors of related poses, every vertex in the pose p_k has same distance γ_k resulting in the same weight $r_k(p_a)$ in every vertex of the pose p_k. Furthermore, because each element of the joint vector equally contributes to the distance calculation, two vectors having a same value but different order generate same pose distance. For example, three different joint vectors $p_1 = (\theta, 0, 0), p_2 = (0, \theta, 0), p_3 = (0, 0, \theta)$ have same distance between them and it can cause unexpected results in PSD.

In WPSD [KM04], Kurihara et al. modify the distance definition between poses using joint weight of each vertex i to give proper weight to each element of a joint vector,

$$\gamma_{i,k}(p_a, p_k) = \sqrt{\sum_{j=1}^{n_{joint}} w_{i,j}(p_{a,j} - p_{k,j})^2} \tag{8}$$

where $\gamma_{i,k}(p_a, p_k)$ is the distance between pose p_a and p_k of vertex i, and $w_{i,j}$ is weight of joint j of vertex i used in equation 2. From this definition, a more accurate pose distance is obtained and it generates better skinning in arbitrary poses, especially when the poses are far from the examples.

Figure 1 shows result of three different skinning methods, but we entrust the detail comparison between quality of each algorithm to their original papers [MTLT88, LCF00, KM04].

4. Computing joint weights from samples

The joint weights of each vertex are important to generate accurate skinning in SSD (equation 2) as well as in WPSD (equation 8). In many applications, the weights are manually generated by skilled artists and it is hard to generate accurate values when a number of joints are involved in deforming a region. In this paper, we automatically calculate the joint weights of each vertex from the sample poses to enhance the accuracy of the weight value. This results in better skinning and reduces the elaborate manual work required to create weight maps.

In each sample pose p_k, we have following equation based on SSD:

$$\tilde{v}_k - e_k = (\sum_{j=1}^{n_{joint}} w_j T_j) v_0 \tag{9}$$

where $\tilde{v}_k$ is a particular vertex from skin sample k, the right hand side is the SSD deformation of vertex v_0 from the rest pose, e_k is a displacement between the SSD deformation and $\tilde{v}_k$, and the other variables are as in equation 2.

If we have sufficient examples involving the same set of n_{joint} joints, we have n_{pose} equations of the form:

$$\tilde{v}_k - e_k = (\sum_{j=1}^{n_{joint}} v_j w_j) \tag{10}$$

where v_j is v_0 transformed by T_j. Although the e_k is unknown, we can solve for weights that minimize the e_k in a least squares sense by stacking the equations 10 (with e_k omitted) into a linear matrix system

$$\|\mathbf{v} - \mathbf{A}\mathbf{w}\|^2 \tag{11}$$

where $\mathbf{w}$ is a n_{joint}-dimensional weight vector, $\mathbf{v}$ is a $3n_{pose}$-dimensional vector containing the vertex $\tilde{v}_i$ from every sample, and $\mathbf{A}$ is a $3n_{pose} \times n_{joint}$ matrix .

>From equation 11, we can calculate $\mathbf{w}$ from the given value of $\mathbf{v}$ and $\mathbf{A}$ to reduce the error of this equation. We use the non-negative least square (NNLS) method to solve this problem and it determines positive weight values minimizing error in equation 10. The calculated weight vector $\mathbf{w}$ is normalized to satisfy $\sum_{j=1}^{n_{joint}} w_j = 1.0$.

In order to avoid a singular matrix A, the number of poses should be greater or equal to the number of overall DOF (Degree Of Freedom) of the joint vector (each joint has 3 DOF), and the sample poses should be sufficiently different.

James et al. used a similar approach to estimate vertex weights in each joint [JT05] and we demonstrate their efforts in our skinning method.

5. Parallel deformation on GPU

Skinning deformations vary across vertices. In SSD and WPSD, this per-vertex computation is independent for each vertex and can be parallelized by a SIMD parallel architecture. We developed a parallel skinning algorithm for SSD and WPSD that is suitable to GPUs having a SIMD architecture with one-side communication to texture memory.

5.1. Parallel WPSD

The computation cost of the SSD skinning algorithm is $O(n_{vertex} \times n_{joint})$ from equations 1, 2, PSD is $O(n_{vertex} \times n_{joint} \times n_{pose})$ from equations 1, 2, 4, and WPSD is $O(n_{vertex} \times n_{joint} \times n_{pose} \times n_{pose} \times n_{pose})$ from equations 1, 2, 4, 5, 6. Where, computation cost of original PSD is defined by equation 1, 2, 4, since r_i is same in all vertices and d_i can be pre-calculated.

The number of joints n_{joint} and poses n_{pose} can be reduced to the smaller numbers using the method developed by Kry et al. [KJP02], as will be discussed in section 5.2.1 with efforts to reduce texture memory space.

In previous research, the Eigenskin method based on PSD was developed using GPU vertex programming [KJP02]. The vertex program uses a relatively small number of slow processing units compared with the fragment processors, and the per-vertex computation cost of the original PSD is $O(n_{joint} \times n_{pose})$. Therefore WPSD, having higher per-vertex computation cost $O(n_{joint} \times n_{pose} \times n_{pose} \times n_{pose})$, can clearly benefit from parallel computation on fragment processors.

5.2. Parallel WPSD on GPU

We developed parallel skinning using the GPU fragment processors and demonstrate our method using three rendering passes. In order to minimize real-time computation, we separate possible pre-calculation steps and save the results into texture memory using texture maps. Because the value in the texture memory is not changed in the successive deformation, it can be pre-computed and stored in the read-only texture memory.

In the first and second pass, per-vertex deformation is calculated in the fragment program and the results are stored in texture maps using the FBO (Frame Buffer Object) extension [Gre05]. These texture maps are bound to the geometry of the rest pose with their texture coordinates. In the third pass, each vertex in the rest pose is changed by the deformed vertex stored in the output texture generated in the first and second passes using vertex texture fetch.

5.2.1. Packing data into textures

The fragment processors cannot access vertex information. Instead, we can use texture memory to send data to the fragment program. Information needed in the fragment program is packed into texture maps and stored into texture memory.

Geometry information from the rest pose is stored into two RGB texture maps, a vertex texture T_v and normal texture T_n; each has size $n_{vertex} \times 3$. These textures represent parameter v_0 in equation 2 and each 3D element (x, y, z) is stored into the (r, g, b) value of a texel [Figure 2].

The joint weights calculated in section 4 are also stored in the texture maps. In general, the distribution of skinning effects in an articulated body is local to several joints [MMT97, KJP02], even in a region as complicated as a hand. For example, deformations arising from the PIP (Proximal Interphalangeal) joint of index finger do not propagate to the other fingers, and deformation on the middle phalanx of index finger is only affected by the movement of PIP and DIP(Distal phalanx) joints. From this observation, we can reduce joint weight storage from the actual number of joint n_{joint} to a smaller number of "principal joints" $\tilde{n}_{joint}$ selected by sorting on the weight value. We threshold $\tilde{n}_{joint}$ at four in our tests with an additional four elements to hold the related joint index. As a result, we can save the joint weights of entire geometry in two RGBA textures T_{w1}, T_{w2} each with size $n_{vertex} \times 4(rgba)$ and store the entire information required for SSD [equation 2] in four textures T_v, T_n, T_{w1}, and T_{w2}.

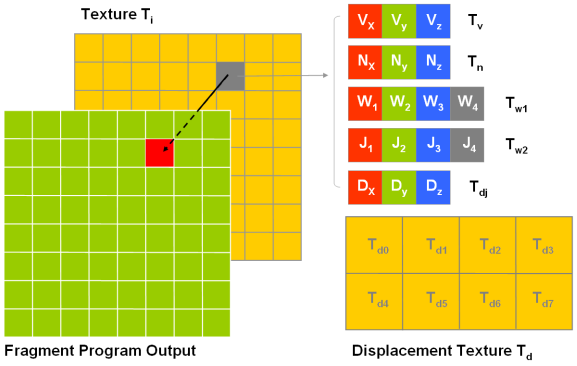

Figure 2: *Packing data into textures: texture map T_i can be T_v, T_n, T_{w1}, T_{w2}, and T_{dj}. V(vertex), N(normal), W(weight), J(joint index), and D(displacement) represent each texel (RGB(A)) value of the related texture. T_d consist of eight T_{dj} storing displacements of each pose j.*

The displacement values calculated by equation 3 can be stored in n_{pose} displacement textures; n_{pose} is the number of sample poses. In case of complex joint structures and a large DOF model, we need many sample poses to calculate accurate joint weights and PSD deformation. However, since the joint weights can be pre-calculated, we can reduce the number of sample poses needed in real-time PSD computation. PCA (Principal Component Analysis) of pose space can yield an orthogonal basis called "Eigendisplacement" [KJP02]. If we reduce the size of pose space from n_{pose} to $\tilde{n}_{pose}$ "principal poses" ($\tilde{n}_{pose} < n_{pose}$), we can reduce the number of displacement textures. In our paper, we set $\tilde{n}_{pose}$ as eight in our experiment and save displacements of all poses into a RGB texture T_d having size $n_{vertex} \times 8(\tilde{n}_{pose}) \times 3(rgb)$.

Therefore, from the two important observations of "principal joints" and "principal poses", the original computation

cost for SSD, PSD, and WPSD discussed in section 5.1 can be reduced using $\tilde{n}_{joint}$ and $\tilde{n}_{pose}$ rather than n_{joint} and n_{pose}.

In the original PSD, since the weight r_i in equation 4 is the same at every vertex, we do not need to calculate this value in the GPU. Since the size of this value is just $\tilde{n}_{pose}$, we can simply pass them to the GPU as parameters without generating a texture map. Therefore, we store all the information needed to calculate the original PSD at this point.

In order to reduce real-time computation, we pre-calculate T_j in equation 2 and λ in equation 5 and store them into another one channel texture T_x having size $\tilde{n}_{pose} \times (\tilde{n}_{pose} + \tilde{n}_{joint} \times 3(x,y,z))$.

As a result, we store all the variables required to calculate WPSD, PSD, and SSD in six texture maps: $T_v, T_n, T_{w1}, T_{w2}, T_d$, and T_x. The values in the texture maps are stored in the texture memory at setup time, since they are not changed during the deformation process.

In current graphic card architectures, data transfer from CPU to GPU is slow compared with memory access within the GPU. Since the only data changed in each deformation and passed from CPU to GPU is a joint vector p_a (size = n_{joint}) representing the current arbitrary pose, the memory access rate in our method is very efficient; In the original PSD method, an additional r_k value (size = $\tilde{n}_{pose}$) is required.

5.2.2. Configurations for fragment program

Variables: T_{out} = output texture, T_v = vertex texture

```
/* Set orthographic camera with same size of quad */;
    gluOrtho2D(-1, 1, -1, 1);
    bind(FBO);
/* Bind Tout and set to FBO drawbuffer */;
    bind(Tout), SetFBOdrawbuffer(Tout);
    bind(Tv);
    enable(fragment program);
/* Set viewport to the resolution of the texture */;
    glViewport(0, 0, texWidth, texHeight);
/* Render a quad into Tout using FBO */;
     glBegin(GL_QUADS);
     {  glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
        glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
        glTexCoord2f(1, 1); glVertex3f( 1, 1, -0.5f);
        glTexCoord2f(0, 1); glVertex3f(-1, 1, -0.5f);
    };
     disable(fragment program);
```

Algorithm 1: Configuration of fragment program for vertex refering and direct rendering into texture

Since vertex information cannot be accessed by the fragment program, vertex deformation on a GPU is usually performed by a vertex program [KJP02, BK05]. Although, we cannot access vertex data in the fragment program, the efficiency of parallel computation on a fragment program is higher, since the fragment processor has more processing units and each of them has more computation power than a vertex processor. The fragment processing system is a general SIMD architecture using fragment streams as input data; each fragment is assigned to a fragment processor to calculate its final color value independently and in parallel.

We developed a parallel WPSD algorithms using the fragment processors to enhance the extent of parallel computation. Geometry information like vertex positions and normals are stored in texture maps T_v and T_n as described in section 5.2.1 and the vertex information is referred in the fragment processors to calculate final color values.

In order to assign each vertex value stored in a texture map to a fragment, we bind the geometry texture T_v or T_n to a quad and render it using an orthographic camera having the same width and height as the quad. Furthermore, since the viewport is set to the same resolution as the textures, each fragment is exactly matched with each texel holding the vertex information, and we can access each vertex using the texture coordinates of the fragment; vertex weights and displacements stored in the texture maps can also be accessed by similar methods.

A similar idea was developed in [PBMH02] to calculate ray tracing in a fragment program and is used in GPGPU (General Purpose computation on GPUs) applications [GPG, LHK*04, PF05].

The FBO (Frame Buffer Object) extension [Gre05] supports rendering into an attached texture. This saves memory and time, since there is no copy operation from frame buffer to texture buffer. We implemented our WPSD algorithm using the fragment program with the FBO extension to store the result directly into texture maps accessed by vertex program in the next pass. A summary of this method is shown in Algorithm 1.

5.3. GPU implementation

We implemented GPU deformation using three rendering passes, and the basic architecture is described in figure 3.

In the first pass, we parallelize per-vertex deformation using GPU fragment processors. The data required to calculate this deformation is stored in the textures as described in section 5.2.1 and the deformation for each vertex is calculated in a fragment processor. In a given arbitrary pose defined by a joint vector, SSD is computed by equation 2 using texture maps T_v, T_{w1}, T_{w2} and T_x; refer to the texture map notation in section 5.2.1. PSD is computed by equation 4 using T_d, T_x, after calculating $r_k(p_a)$ by equation 6. In the case of WPSD, the weighted distance is computed by equation 8 using T_{w1}, T_{w2}, and T_x.

In the first pass, the result of the deformation is rendered

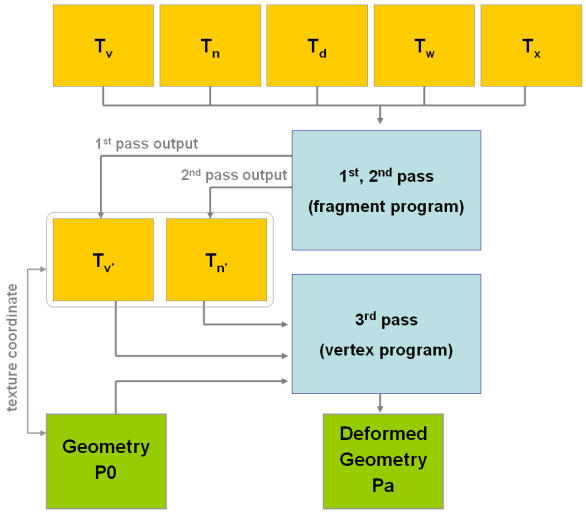

Figure 3: *Overview of WPSD on GPU: Each T_i is the texture map storing the required data for the calculation (refer to section 5.2.1 for their values) , $T_{i'}$s are the texture maps to store the output of the 1st and 2nd passes, P_0 is the geometry in the rest pose, and P_a is a deformed model in an arbitrary pose.*

into a texture map $T_{v'}$, using the FBO, and passed to the third pass. In the second pass we calculate and store normal deformation with a similar method as in the first pass, and the results are stored in the texture map $T_{n'}$.

In the third pass, using a vertex program, each vertex of the rest pose is transformed to the final deformed position using the information from the texture maps computed in the previous two passes. In order to access related texture information in each vertex, we created texture coordinates of each texel in pre-processing and used them in the vertex program. Specifically, the two texture maps, $T_{v'}$ and $T_{n'}$ that are generated in the first and second passes are accessed in the vertex program using the texture coordinate of the current vertex.

Alternatively, multiple render targets (MRTs) can combine the first and second pass, and vertex buffer objects (VBOs) could be used to render the deformed results back to the vertex array [OPE, GPG, LHK*04].

6. Results

We tested our methods using upper arm models consisting of four joints (collar, shoulder, elbow, and wrist). Each has three DOF and the wrist is the end joint having no DOF. Three different resolution meshes are used to test the performance of GPU parallel computation: the high-resolution model has 91460 triangles with 46036 vertices, the mid-resolution model has 22836 triangles with 11574 vertices, and the low-resolution model has 5762 triangles with 2972 vertices [Figure 4]. Note that these models are considerably more detailed than those used in current games, so the reported frame rates would be much higher if typical game-resolution models were used. On the other hand, with the expected growth of GPU processing power, models such as these will be in wide use in a few years, and algorithms such as WPSD will be required to produce realistic deformations at this level of resolution.

Figure 4: *Mesh of test data: the top row is a low-resolution mesh, the second row is a mid-resolution mesh, and the bottom row is a high-resolution mesh*

Eight sample poses were created by Poser [Cur] and the joints weights and displacements of each sample were derived from these models [Figure 5].

Our parallel algorithm is based on three pass GPU computation. The fragment program for the 1st and 2nd pass, and the vertex program for the 3rd pass are implemented in the Cg language [FK03]. For accuracy the GPU computation is performed by 32bit floating point operations with 32bit floating point texture maps. Table 2 shows the total memory space to store texture maps required by the fragment program. Note that the maximum required memory space for the highest resolution model is just 6.8 Mbytes; the size of the output texture $T_{v'}$ and $T_{n'}$ is the same as the size of T_v and T_n.

The results of GPU-based deformation for SSD, PSD, and WPSD are shown in Figure 1 and 6, and the experiment is performed in a GeForce 6800 Ultra GPU and a 3.4Ghz Pentium 4 CPU. The timing results of each algorithm on the CPU and GPU are summarized in table 1.

On average, our GPU-based deformation shows around 20 times speed-up compared with CPU-based deformation. GPU-based WPSD has roughly the same speed as CPU-based SSD. Therefore, real-time applications using SSD can substitute WPSD running on the GPU without loosing their real-time performance. Since our algorithm shows speed-up for SSD and PSD as well as WPSD, applications can choose the most appropriate skinning method according to the required deformation and detail.

Method	Mesh	CPU(FPS)	GPU(FPS)
SSD	low	150	1425
	middle	39	630
	high	5	164
PSD	low	98	1230
	middle	23	530
	high	4.5	140
WPSD	low	5	85
	middle	1.2	25
	high	0.29	7

Table 1: *Timing results (in FPS) of each algorithm: the low-resolution mesh has 5762 triangles with 2972 vertices, the mid-resolution mesh has 22836 triangles with 11574 vertices, and the high-resolution mesh has 91460 triangles with 46036 vertices.*

Vertices	T_v&T_n	T_{w1}&T_{w2}	T_d	T_x	Tot
2972 (low)	35×2	46×2	278	1	441
11574 (mid)	135×2	180×2	1080	1	1711
46036 (high)	539×2	719×2	4315	1	6832

Table 2: *Texture memory to store data required in fragment program (in Kbytes); refer to section 5.2.1 for texture notation .*

7. Conclusions

In this paper, we present a parallel skinning algorithm suitable for SIMD architectures such as GPUs. The joint weights of each vertex are automatically computed by NNLS and used in the skinning computation for SSD and WPSD.

Independent per-vertex deformation is parallelized on the GPU using three rendering passes. In the first and second passes, per-vertex deformation is calculated by the fragment processors and the results are stored in texture maps using FBO. In the third pass, using vertex processors, each vertex of the rest pose is changed by the deformed vertex stored in the textures generated by the first and second passes.

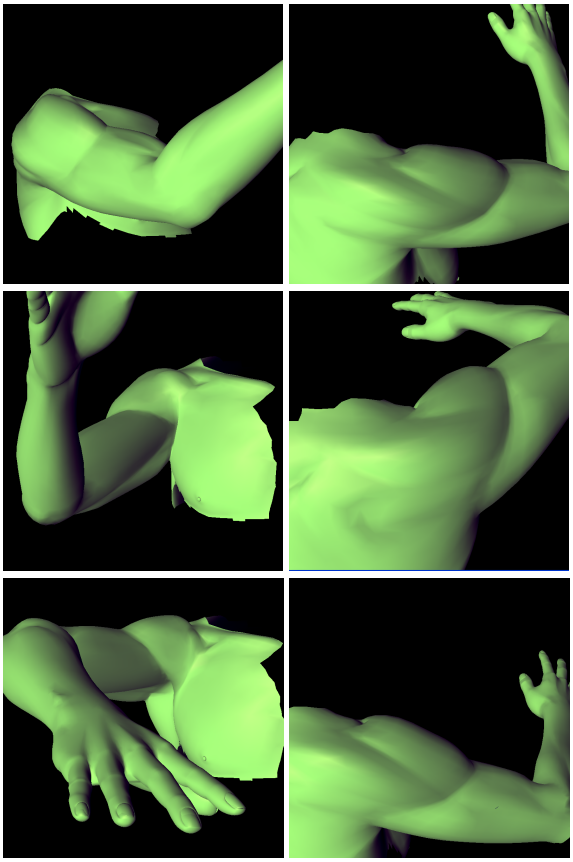

Figure 6: *Arbitrary poses deformed by WPSD on GPU*

Articulated body skinning using SSD, PSD, and WPSD are efficiently parallelized by our GPU-based method, and on a detailed model, we obtain around 20 times speed-up compared with CPU-based computation.

Principal component compression of the examples and careful analysis of joint distributions can reduce the domain of computation [KJP02] and other algorithms based on the SSD, PSD, and shape interpolation may be parallelized on GPU using our approach.

Acknowledgments

This research has been funded by the Integrated Media System Center/USC, and Samsung Electronics. We wish to thank KyungKook Park, Changki Min, and Tim Foley for discussions about GPUs, and the anonymous reviewers for their sincere comments.

References

[ACP02] ALLEN B., CURLESS B., POPOVIĆ; Z.: Articulated body deformation from range scan data. In *SIGGRAPH '02: Proceedings of the 29th annual conference*

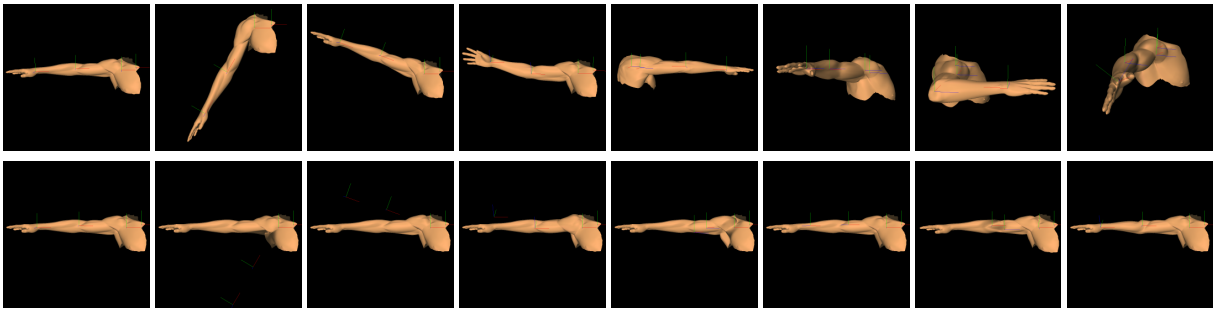

Figure 5: *Samples poses and displacements: the first row shows each sample poses, the second row shows displacement of each sample pose with the rest pose in the first column of the third row. Please enlarge to see details.*

on Computer graphics and interactive techniques (New York, NY, USA, 2002), ACM Press, pp. 612–619.

[AHS03] ALBRECHT I., HABER J., SEIDEL H. P.: Construction and animation of anatomically based human hand models. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA-03)* (2003), pp. 98–109.

[ASK*05] ANGUELOV D., SRINIVASAN P., KOLLER D., THRUN S., RODGERS J., DAVIS J.: Scape: shape completion and animation of people. *ACM Trans. Graph. 24*, 3 (2005), 408–416.

[BK05] BOTSCH M., KOBBELT L.: Real-time shape editing using radial basis functions. *Computer Graphics Forum 24*, 3 (2005), 611–621. (Proceedings of Eurographics 2005).

[CBC*05] CAPELL S., BURKHART M., CURLESS B., DUCHAMP T., POPOVIĆ; Z.: Physically based rigging for deformable characters. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2005), ACM Press, pp. 301–310.

[Cur] CURIOUSLAB: Poser 6. `http://www.curiouslabs.com`.

[FK03] FERNANDO R., KILGARD M. J.: *The Cg Tutorial; The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[GPG] GPGPU: General-purpose computation using graphics hardware. `http://gpgpu.org`.

[Gre05] GREEN S.: *The OpenGL Framebuffer Object Extension*. Game Developers Conference, 2005. `http://developer.nvidia.com/object/gdc_2005_presentations.html`.

[JP02] JAMES D. L., PAI D. K.: Dyrt: dynamic response textures for real time deformation simulation with graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 582–585.

[JT05] JAMES D. L., TWIGG C. D.: Skinning mesh animations. *ACM Trans. Graph. 24*, 3 (2005), 399–407.

[KJP02] KRY P. G., JAMES D. L., PAI D. K.: EigenSkin: Real time large deformation character skinning in hardware. In *Proceedings of the 2002 ACM SIGGRAPH Symposium on Computer Animation (SCA-02)* (2002), pp. 153–160.

[KM04] KURIHARA T., MIYATA N.: Modeling deformable human hands from medical images. In *Proceedings of the 2004 ACM SIGGRAPH Symposium on Computer Animation (SCA-04)* (2004), pp. 357–366.

[KZ05] KAVAN L., ZARA J.: Spherical blend skinning: A real-time deformation of articulated models. In *2005 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (April 2005), ACM Press, pp. 9–16.

[LCF00] LEWIS J. P., CORDNER M., FONG N.: Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 165–172.

[LHK*04] LUEBKE D., HARRIS M., KRUGER J., PURCELL T., GOVINDARAJU N., BUCK I., WOOLLEY C., LEFOHN A.: Gpgpu: general purpose computation on graphics hardware. In *GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes* (New York, NY, USA, 2004), ACM Press, p. 33.

[MMT97] MOCCOZET L., MAGNENAT-THALMANN N.: Dirichlet free-form deformations and their application to hand simulation. In *Computer Animation* (1997).

[MTG03] MOHR A., TOKHEIM L., GLEICHER M.: Direct manipulation of interactive character skins. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics* (New York, NY, USA, 2003), ACM Press, pp. 27–30.

[MTLT88] MAGNENAT-THALMANN N., LAPERRIÈRE R., THALMANN D.: Joint–dependent local deformations for hand animation and object grasping. In *Graphics Interface '88* (June 1988), pp. 26–33.

[OPE] OPENGL: Opengl extension registry. `http://oss.sgi.com/projects/ogl-sample/registry/`.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[PCLS05] PRATSCHER M., COLEMAN P., LASZLO J., SINGH K.: Outside-in anatomy based character rigging. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2005), ACM Press, pp. 329–338.

[PF05] PHARR M., FERNANDO R.: *GPU Gems 2; Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[RNL06] RHEE T., NEUMANN U., LEWIS J.: Human hand modeling from surface anatomy. In *I3DG '06: Proc. of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2006).

[SNF05] SIFAKIS E., NEVEROV I., FEDKIW R.: Automatic determination of facial muscle activations from sparse motion capture marker data. *ACM Trans. Graph. 24*, 3 (2005), 417–425.

[SRC01] SLOAN P.-P. J., ROSE C. F., COHEN M. F.: Shape by example. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (New York, NY, USA, 2001), ACM Press, pp. 135–143.

[Web00] WEBER J.: Run-time skin deformation. In *In Proceedings of Game Developers Conference* (2000).

[WP02] WANG X. C., PHILLIPS C.: Multi-weight enveloping: least-squares approximation techniques for skin animation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2002), ACM Press, pp. 129–138.