Virtual Bones: animating skinned meshes on GPU

Vladimir V. Lopatin[*]
Senior VFX Artist, Guerrilla-Games

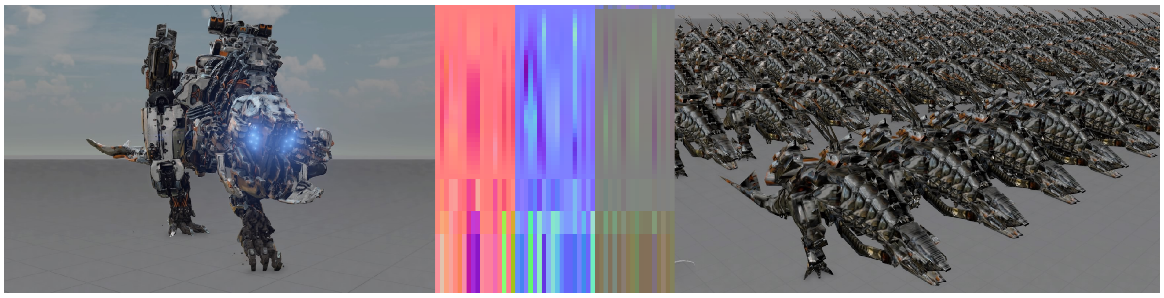

Figure 1: *A skinned mesh character, a virtual bones texture and multiple 3D impostors.*

Abstract

During the production of open-world game Horizon: Zero Dawn, we had to find a way to render large amounts of animated assets in real-time, such as crowds and locations heavily decorated with animated cloth-like assets. Analysis showed that existing assets are CPU-heavy and are resisting scaling to large numbers. Instead of using skinned meshes, driven by skeletons with complex CPU controll structures, assets had to become vastly simpler and computations had to be offset to GPU. That required creating a new pipeline that includes generating and authoring new content. We wanted to reuse existing content, so it had to be compatible with existing pipeline. We already had an implementation of [Norman], which allowed us creating assets with animationions baked into a texture that drives a vertex displacement program on GPU. However, [Norman] approach is vertex-count bound, which meant that for large meshes the animation texture size had to be large. Because our assets already had skeleton, animation and skinning data, we decided to develop a method that would do a form of skinning on GPU, but would be bound by bones, instead of vertices as in [Norman], and use existing skinning data. These challenges had to be met when programmers support time is very limited. Instead, we decided to use SideFX Houdini Animation Tools to set up a VFX pipeline to process existing assets and use existing shading programs in a version of Autodesk Maya, which did not require any extra work from programmers. This allowed us to keep animation texture size small, offset nearly all computations to GPU while keeping CPU footprint minimal or nill; we could reuse existing content, it did not require changing existing pipeline or code support from programmers, while significanly improving the visual results by allowing us to render a large amount of animated assets.

Keywords: GPU, crowd-rendering, shading, skinned meshes.

Concepts: •Computing methodologies → Image manipulation; *Computational photography;*

1 Previous Work

Conceptually similar ideas have been expressed before, among others are: texture-driven vertex displacement by [Norman 2015], [ratchet et al.].

[*]e-mail:vladimir.lopatin82@gmail.com

We already had an implementation of [Norman], which allowed us creating assets with animations baked to a texture that drives a vertex displacement program on GPU. However, this approach is vertex-count bound, i.e. vertex count = horizontal texture resolution; it requires additional normal transformation texture. We had to look for a method to keep the texture size small.

2 Implementation

The proposed method includes a vertex displacement shader, driven by a texture. The texture is a rows-columns lookup table, where every row of pixels corresponds to a key-frame and every column pixel encodes a component vector of a transofrm 4×4 matrix. A component vector is a 3 component vector, corresponding to RGB, hence 2 vectors can be encoded as RGBRGB (6 color channels or 2 RGB pixels). If we chose to use quaternions to store rotation, we could have used RGBA (4 channels or 1 RGBA pixel instead), thus saving 30% of horizontal texture resolution, but implementing matrix transformation is trivial and quaternion optimization is left for future work.

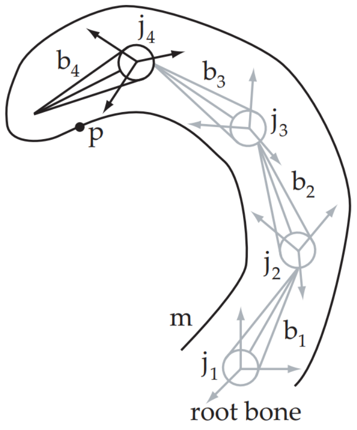

Figure 2: *A skin mesh, m, being deformed by a skeleton consisting of bones, b_i , connected by single joints. Joint j_i is a transform belonging to bone b_i . The vertex, p, is influenced by the one or more bones b_i.*

A bone has two transforms; a bone space transform, B_j^{-1} , and a pose transform, P_j . Both transforms can be represented as 4×4

homogeneous transformation matrices (figure 1).

$$T_j = \begin{pmatrix} T_j^{rot} & T_j^{trans} \\ \bar{0} & 1 \end{pmatrix} \quad (1)$$

where T_j^{rot} is a 3×3 rotation matrix, $\bar{0}$ is a 1×3 zero vector, and T_j^{trans} is a 3×1 vector. The transform can also be represented as (quaternion, transform) pairs, which is more compact. Since we are not trying to preserve the skeleton relationships and are only interested in the influence of a set of bones on a vertex position and normal orientation, we discard the relationships that have no influence and treat each joint orientation and position as a transfomation 4×4 matrix. Then we perform a linear blend between a set of such transformation matrixes to get a final transformation matrix that we apply to a vertex. The linear blend is controlled by weights. Weights can be stored as a texture or as a secondary uv-set as proposed in [Norman]. We do that for every vertex in a vertex displacement shader, hence a vertex position can be computed as:

$$\bar{P}_i = \sum_{j=0}^{n} ((\bar{p}_i^{rest} - \bar{t}_j^{trans}) \cdot T_j^{-1} + \bar{t}_j^{trans}) \cdot \omega_j \quad (2)$$

where $\bar{P}_i$ is the resulting vertex position, $\bar{p}_i^{rest}$ - rest vertex position and n - number of bones that influence i^{th} vertex, $\bar{t}_j^{trans}$ is a rest-pose translation of j^{th} joint and ω_j is it's respective weight.

$$\bar{N}_i = \sum_{j=0}^{n} (\bar{N}_i^{rest} \cdot T_j^{-1}) \cdot \omega_j \quad (3)$$

where $\bar{N}_i$ - normal of i^{th} vertex. As we can see, (3) and (2) are nearly identical except for translation.

We store T_j and T_j^{rest} as a reduced matrix (we store 2 rotation vector components and restore the 3rd as a cross product of the two). The joints rest position and rotation is stored in the first row of pixels, hence it's deemed a technical row and is not a part of animation loop.

Vertex rest positon $\bar{p}_i^{rest}$ and rest normal $\bar{N}_i^{rest}$ are supplied by the rest-mesh. Bone indices that influence each vertex correspond to horizontal pixel offset in the texture and are saved as vertex colors in the mesh (Figure 4). By using 4 8-bit vertex colors (RGBA), we can map up to 256 bones with 4 influences per vertex, resulting in a form of smooth-skinning, or use a single channel, resulting in hard-skinning. In other words we store a bone number (or, equivalently, a horizontal pixel offset) as a vertex color and later read it in a shader in order to drive the horizontal offset of a texture lookup (Figure 4)

Let $\bar{\theta}, \bar{\phi}, \bar{\zeta}$ be the vector components of the rotation matrix T_j^{rot}:

$$[T_j^{rot}]^T = \begin{pmatrix} \theta_1 & \theta_2 & \theta_3 \\ \phi_1 & \phi_2 & \phi_3 \\ \zeta_1 & \zeta_2 & \zeta_3 \end{pmatrix} \quad (4)$$

and let *texture(u,v)* be a function, that, given *(u,v)*, where $\{u, v \in \mathbf{R}; u, v \in \{0, 1\}\}$, returns a pixel (RGB) from a texture, then, since

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} = \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (5)$$

$$\begin{aligned} \theta_j &= texutre[Cd.j_i \cdot 256 \cdot n \cdot C), t] \\ \phi_j &= texutre[(Cd.j_i \cdot 256 \cdot n + n) \cdot C), t] \\ \zeta_j &= \theta_j \cdot \phi_j \\ T^{trans} &= texutre[(Cd.j_i \cdot 256 \cdot n + 2 \cdot n) \cdot C), t] \end{aligned} \quad (6)$$

where n - number of bones, $t \in \mathbf{R}$ - animation offset (time), $Cd.j_i$ is a vertex color, j^{th} component (RGBA) of a vertex i, multiplication by 256 is necessary to map $\{$0-255$\}$ (bone index) to $\{$0-1$\}$ (8bit vertex color value), $C \in \mathbf{R}$ is a scalar space-normalisation constant

The resulting horizontal texture resolution is $n \cdot 3$, rounded to the closes power of 2, vertical resolutoin equals the number of frames (Figure 3).

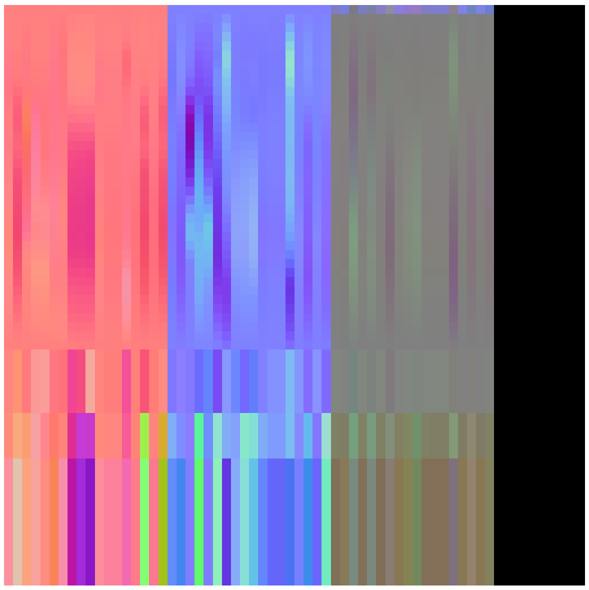

Figure 3: *The top-row of pixels stores joints rest rotation and position. Columns 1 (red) and 2 (blue) encode animation of joint rotation and position (column 3). Black area to the right is unused space. Horizontal bands in lower-half part of the texture are animation loops and poses.*

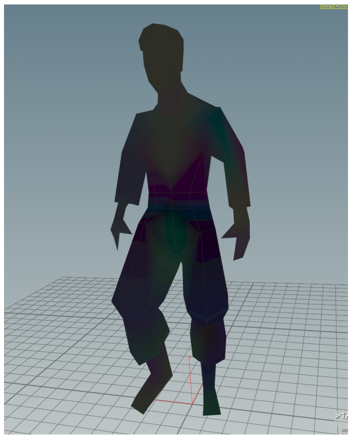

Figure 4: *A rest-mesh with vertex colors encoding bone indices.*

As we have mentioned before, weights can be stored in a number of ways. We assume that the sum of weights per vertex equals 1:

$$\sum_{j=0}^{n} \omega_j = 1 \tag{7}$$

Hence, in order to store 3 bone weights, we assume:

$$\omega_1 + \omega_2 + \omega_3 = 1$$
$$\omega_3 = 1 - (\omega_1 + \omega_2)$$

Therefore, we only need to store two components to deduce the third, similarly with other number of bone weights.

Because the resulting size is only $\sqrt{n}$, where n is the number of vertexes, which tends to be a small number when dealing with low-res meshes, and also because it seemed like a good idea at the moment, we decided to use a secondary texture and a secondary uv-set (uv2) to store and map bone weights (Figure 5):

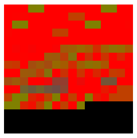

Figure 5: *16x16 texture used to store bone wieghts*

By using the virtual bones texutre (Figure 3) we can build a transformation matrix that, when applied to rest-mesh vertex position and vertex normal, returns a new vertex position and normal. By scrolling the texture lookup vertically, we can achieve animation.

2.1 Animation switching

Figure 3 is demonstrating a number of animaion loops encoded in a single texture: a number of animation slices are visible in the lower half of the image. In order to switch animations, the bone animation texture can store a necessary numbder of loops. The animation lookup can be limited to a sertain texture area, thus achieving animation switching. That can be set during initialisation at spawn time of the object, or using more sophisticated logic.

2.2 Texture generation

The method is supporting generating the virtual bones animation texture using either existing skeleton or can convert procedural joints and other sticky objects, treating them as joints by converting their transformation data into a texture. The authors used SideFX Houdini Amimation Tools to read the Maya-exported fbx file, containing the rig and animation data, and encode the joints transformations into a texture.

3 Results and Performance.

The following (table 1) contains performance cost of 1 to 100 instances of test object (base-line (BL)) and its respective 3D impostor instances (1-1200) (virtual bones (VB)). By ‘baseline algorithm‘ (BL) we understand a skeleton-based CPU-driven skinned mesh, in this specific case it’s a character from HZD. Frames per second count is capped at 30 fps. BL is using the same amount of triangles (tris.) as VB. The test is performed running a test level with a dedicated minimalistic setup, using Decima engine.

Table 1: *Performance comparison, whe N - number of instances, tris. - triangles count, fps - frames per second, VB - virtual bones method, BL - baseline method.*

N	tris.	fps		CPU time (μs)		GPU time (%)	
		BL	VB	BL	VB	BL	VB
1	4652	30	30	3168	0	0.317	0.100
2	9304	30	30	4151	0	0.582	0.192
3	13956	30	30	6729	0	0.642	0.281
4	18608	30	30	8554	0	0.704	0.354
5	23260	30	30	9889	0	0.834	0.423
6	27912	30	30	11929	0	1.117	0.501
7	32564	30	30	13098	0	1.141	0.581
8	37216	30	30	14771	0	1.179	0.645
9	41868	30	30	16689	0	1.267	0.706
10	46520	30	30	19400	0	1.385	0.756
20	93040	30	30	29514	0	2.550	1.449
30	139560	30	30	47504	0	3.258	2.170
40	186080	24	30	118988	0	3.974	2.884
50	232600	15	30	153831	0	5.083	3.607
60	279129	13	30	170660	0	5.897	4.178
70	325640	12	30	192778	0	6.709	4.609
80	372160	10	30	226705	0	7.340	5.031
90	418680	9	30	254145	0	8.513	5.580
100	465200	8	30	283474	0	9.305	5.987
200	953400	-	30	-	0	-	11.183
300	1430100	-	30	-	0	-	13.680
400	1906800	-	30	-	0	-	17.443
500	2383500	-	30	-	0	-	21.040
600	2860200	-	30	-	0	-	24.027
700	3336900	-	30	-	0	-	26.736
800	3813600	-	30	-	0	-	29.608
900	4290300	-	30	-	0	-	32.795
1000	4767000	-	30	-	0	-	35.837
1200	5720400	-	30	-	0	-	41.459

By looking at table 1, column ’fps’ (BL), we can observe a noticable FPS drop for basline algorithm at around 30-40 instances due to CPU time cost, where it is becoming prohibitively expsnsive. On the contrary, VB does not show any noticable frame-rate drop, due to VB zero CPU time cost (Figure 6).

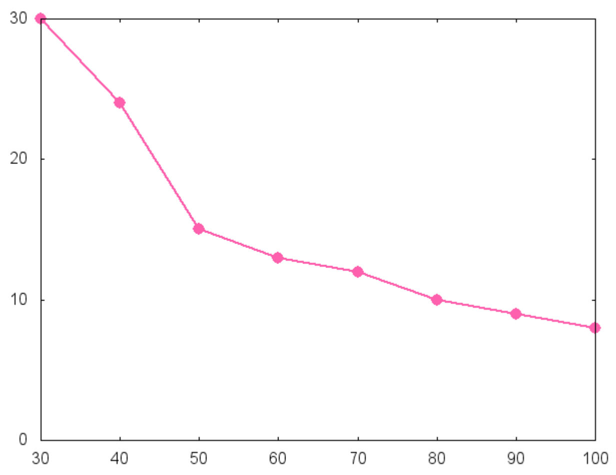

Figure 6: *horizontal axis: number of instances, vertical axis: frames per second. (baseline algorithm)*

Looking at GPU frame time, we can see that BL curve is growing much faster, converging around $O(n)$, while VB graph seems to converge to $O(log(n))$. (Figures 7 and 8)

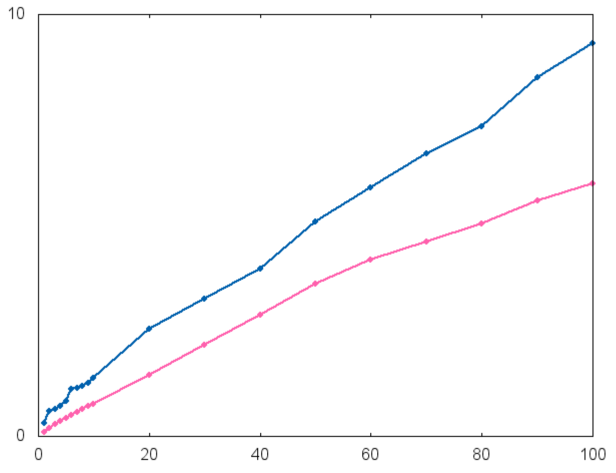

Figure 7: *horizontal axis: number of instances, vertical axis: GPU frame time. BL (blue), VB(red)*

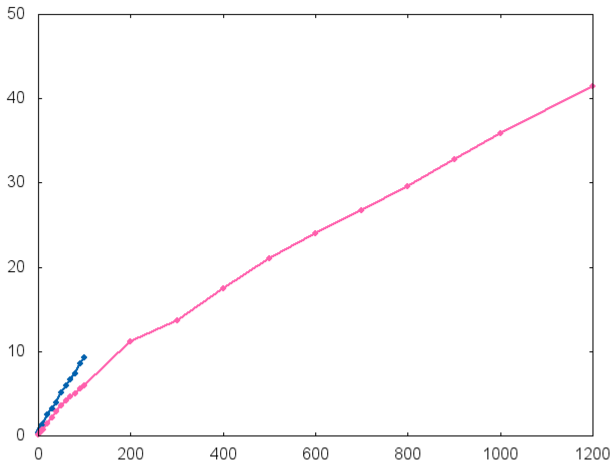

Figure 8: *horizontal axis: number of instances, vertical axis: GPU frame time. BL (blue), VB(red)*

4 Limitations and Future Work

Because the horizonral texture resolution was rounded to the closest power of 2, that led to some texture space waste. We used transformation matrices, instead of quaternions - using quaternions would allow further reducing texture size. In order to minimize rounding errors, the animation texture had to be stored as 32bit uncompressed. Storing rotation and translation components in 2 separate textures may improve packing and reduce space waste. More research is required in regards to losless compression of animation texture.

Acknowledgements

To Marijn, for all the wisky.

References

AGARWAL, S., MIERLE, K., AND OTHERS. Ceres solver. https://code.google.com/p/ceres-solver/.

ANONYMOUS, 1976. Planes of the head. http://www.planesofthehead.com/.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 15–22.

JOBSON, D. J., RAHMAN, Z., AND WOODELL, G. A. 1995. Retinex image processing: Improved fidelity to direct visual observation. In *Proceedings of the IS&T Fourth Color Imaging Conference: Color Science, Systems, and Applications*, vol. 4, The Society for Imaging Science and Technology, 124–125.

KARTCH, D. 2000. *Efficient Rendering and Compression for Full-Parallax Computer-Generated Holographic Stereograms*. PhD thesis, Cornell University.

LANDIS, H., 2002. Global illumination in production. ACM SIGGRAPH 2002 Course #16 Notes, July.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The digital michelangelo project. In *Proceedings of SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH, New York, K. Akeley, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 131–144.

PARK, S. W., LINSEN, L., KREYLOS, O., OWENS, J. D., AND HAMANN, B. 2006. Discrete sibson interpolation. *IEEE Transactions on Visualization and Computer Graphics 12*, 2 (Mar./Apr.), 243–253.

PARKE, F. I., AND WATERS, K. 1996. *Computer Facial Animation*. A. K. Peters.

PELLACINI, F., VIDIMČE, K., LEFOHN, A., MOHR, A., LEONE, M., AND WARREN, J. 2005. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics 24*, 3 (Aug.), 464–470.

SAKO, Y., AND FUJIMURA, K. 2000. Shape similarity by homotropic deformation. *The Visual Computer 16*, 1, 47–61.

YEE, Y. L. H. 2000. *Spatiotemporal sensistivity and visual attention for efficient rendering of dynamic environments*. Master's thesis, Cornell University.