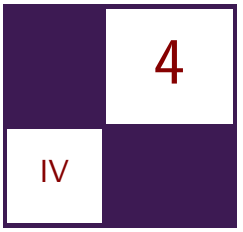

Efficient Morph Target Animation Using OpenGL ES 3.0

James L. Jones

4.1 Introduction

Demand for impressive graphics in mobile apps has given rise to ever-more powerful GPUs and new graphics APIs such as OpenGL ES 3.0. These advances enable programmers to write cleaner and more efficient implementations of computer graphics algorithms than ever before. One particular area that stands to benefit is the facial animation of characters in mobile games, a technique that is commonly implemented using morph targets.

Morph target animation requires artists to pre-create multiple poses of a model offline. Later on, during application execution, these poses are mixed together in varying quantities to create animation sequences such as blinking eyes or a frowning face. When used in conjunction with skinning, the technique can serve as the foundation of a feature-rich character animation system. Historically, morph target animation has seen heavy use in PC and console games but a lack of flexibility in earlier graphics APIs has led to complicated or expensive implementations on mobile platforms. This chapter describes an efficient implementation of morph target animation that makes use of the transform-feedback API introduced in OpenGL ES 3.0. The application in this chapter is a continuation of the Gremlin demo, which was first introduced in a previous article [Senior 09]. The demo has been entirely rewritten and has been updated to take advantage of OpenGL ES 3.0 on PowerVR Series6 mobile GPUs.

4.2 Previous Work

An interesting geometry texturing approach was suggested for mobile GPUs that made use of the OpenGL ES 2.0 API [Senior 09]. This approach used a vertex encoding scheme to store vertex displacements between target poses in textures.

The morphing procedure would be performed entirely on textures bound to framebuffer objects, with the final texture then being used to displace mesh vertices using vertex texture fetch operations. There are problems with this approach, most notably the fact that the maximum number of texture units provided by a platform is allowed to be zero [Munshi and Leech , pp. 40–41]. This means that on some platforms an alternative approach would need to be used. Transform-feedback functionality, which is standardized in OpenGL ES 3.0, allows for a simpler implementation and removes the need for wasteful vertex texture encoding and decoding operations.

4.3 Morph Targets

Morph target animation is used in cases where many small, per-vertex changes need to be applied to a model. This is in contrast to large, sweeping motions normally handled by skeletal animation systems. A good use-case for morph targets is the animation of facial muscles required to create believable facial expressions for video game characters. (See Figure 4.1.)

An implementation typically operates on multiple versions of a stored mesh, known as target poses or key poses. These poses are stored in conjunction with a base pose, which serves as a representation of the animation in a neutral state. To create different animation sequences, the position of each mesh vertex is blended with one or more target poses using a weight vector. The components of this vector are associated with a corresponding target pose and denote how much this target pose influences the result.

To be able to blend between target poses, a difference mesh is used. This is a mesh that is created for each target pose that gives the per-vertex difference between the target pose and the base pose [Senior 09,Lorach 08]. These difference

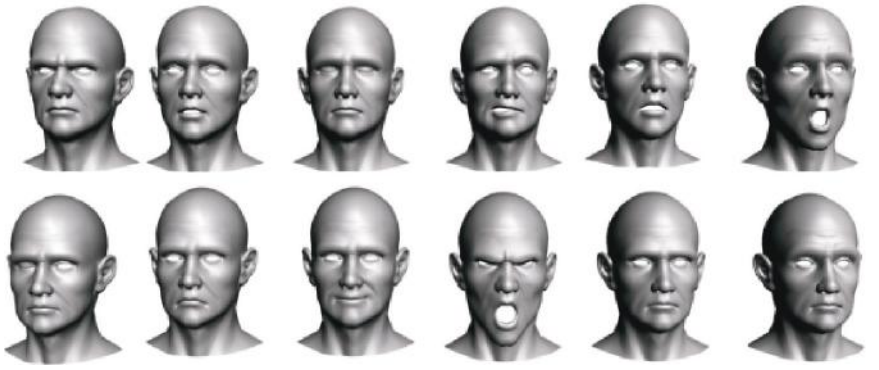

Figure 4.1. Multiple target poses showing a range of facial expressions.

vectors are used as bases in a vector space (i.e., each vertex in an output mesh can be constructed by taking a linear combination of these bases using a weight vector). More precisely, for each output vertex v_i at time t in N morph targets with base target vertex b_i, weight vector w, and target pose vertex p_i, we have

$$\boldsymbol{v_i}(t) = \boldsymbol{b_i} + \sum_{k=0}^{N} w_k(t) \cdot (\boldsymbol{p_{k,i}} - \boldsymbol{b_i}).$$

The formula above summarizes all that is necessary for a morph target implementation. However, for many scenes, it is often the case that only some of the total possible weights change every frame. Because of this, we want to avoid wastefully re-calculating the entire contribution from all target poses every frame. A better idea is to instead keep track of the changes in the weight vector along with the current pose in memory. For a change in frame time h, the new position is equal to the current pose position plus the change in this position:

$$\boldsymbol{v_i}(t+h) = \boldsymbol{v_i}(t) + \Delta_h[\boldsymbol{v_i}](t).$$

We also see that the per-frame change in the position depends only on the per-frame changes in the weights:

$$\Delta_h[\boldsymbol{v_i}](t) = \sum_{k=0}^{N} \Delta_h[w_k](t) \cdot (\boldsymbol{p_{k,i}} - \boldsymbol{b_i}).$$

Using this information, we develop an approach where we only need to compute and update the pose with the per-frame contribution for weights that have changed, i.e., when $\Delta_h[w_k](t) \neq 0$.

4.4 Implementation

This implementation uses vertex buffers that are bound to a transform feedback object to store and update the current pose across frames. Unfortunately, OpenGL ES 3.0 prevents reading and writing to the same buffer simultaneously so a secondary vertex buffer is used to ping-pong the current pose (i.e., the output buffer is swapped with the input buffer every frame). The difference meshes are computed in a pre-processing pass by iterating through the vertices and subtracting the base pose. Sensible starting values are loaded into the feedback vertex buffers (in this case the base pose is used). Every frame, we update the current pose in the vertex buffers using the changes in the weights. This update can be performed with or without batching. Finally, we render the contents of the updated vertex buffer as usual. We perform the computation on vertex normals in the same fashion as vertex positions; this gives us correctly animated normals for rendering. (See Figure 4.2.)

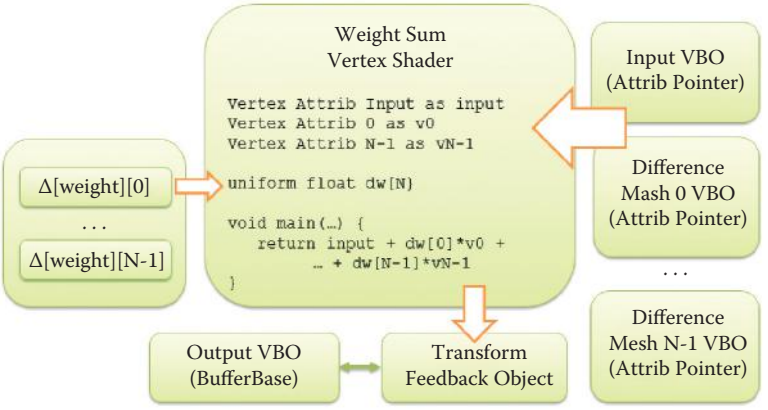

Figure 4.2. Technique overview.

4.4.1 Pose Update

Before we update the pose, we must first compute the changes in the weights. The following pseudo code illustrates how this can be done.

```
// Inputs:
// w[] : current frame weight vector
// p[] : previous frame weight vector
// dw[] : delta weight vector

// Per-Frame Weight Update:
animate(w)
for i = 0 to length(w):
    dw[i] = w[i] - p[i]
    p[i] = w[i]
```

Using the delta weights we can now check to see what components of the weight vector have changed. For those weights that have changed, do a transform feedback pass using the key-pose.

```
// Inputs:
// q[] : array of difference mesh VBOs
// dw[] : delta weight vector
// vbo[] : array of vertex buffer objects for storing current
//         pose
// tfo : transform feedback object
// shader : shader program for accumulation

// Per-Frame Pose Update:
glBindTransformFeedback(tfo)
glEnable(RASTERIZER_DISCARD)
```

```
glUseProgram(shader)
for(i = 0, i < length(q), i++):
    // Only for weights that have changed...
    if (abs(dw[i]) != 0):
        // Set the weight uniform
        glUniform1f(...,dw[i])
        // Bind the output VBO to TBO
        glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,vbo[1])
        // Bind the inputs to vertex shader
        glBindBuffer(GL_ARRAY_BUFFER,vbo[0])
        glVertexAttribPointer(ATTRIBUTE_STREAM_0,...)
        glBindBuffer(GL_ARRAY_BUFFER,q[i])
        glVertexAttribPointer(ATTRIBUTE_STREAM_1,...)
        // Draw call performs per-vertex accumulation in vertex
        // shader.
        glEnableTransformFeedback()
        glDrawArrays(...)
        glDisableTransformFeedback()
        // Vertices for rendering are referenced with vbo[0]
        swap(vbo[0],vbo[1])
```

4.4.2 Batching

For efficiency, further improvements can be made. In this version, updates are batched together into fewer passes. Instead of passing in the attributes and weight uniform for one pose at a time, we can instead use a vertex shader that processes multiple key-poses per update pass. We then perform the update with as few passes as possible.

```
// Inputs:
// q[] : difference mesh VBOs, where corresponding dw != 0
// vbo[] : array of vertex buffer objects for storing current
//         pose
// dw[] : delta weight vector
// shader[] : shader programs for each batch size up to b
// b : max batch size

// Per-Frame Batched Pose Update:
// ... Similar setup as before ...
for(i = 0, i < length(q), ):
    k = min(length(q),i+b) - i
    glUseProgram(shader[k])
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,0,vbo[1])
    // Bind attributes for pass
    for(j = 0, j < b, j++):
        if (j < k):
            glEnableVertexAttribArray(ATTRIBUTE_STREAM_1+j)
            glBindBuffer(GL_ARRAY_BUFFER,q[i+j])
            glVertexAttribPointer(ATTRIBUTE_STREAM_1+j,...)
        else:
            glDisableVertexAttribArray(ATTRIBUTE_STREAM_1+j)
    // Set the delta weights
    glUniform1fv(...)
    // Bind current pose as input and draw
    glEnableVertexAttribArray(ATTRIBUTE_STREAM_0)
    glBindBuffer(GL_ARRAY_BUFFER,vbo[0])
```

```
glVertexAttribPointer(ATTRIBUTE_STREAM_0,...)
glEnableTransformFeedback()
glDrawArrays(...)
glDisableTransformFeedback()
swap(vbo[0],vbo[1])
i = i + k
```

This technique requires multiple versions of the shader where each version executes the summation on incrementally more input attributes up until the maximum batch size. It is important to note that the maximum number of input attributes available is limited by the API. This value (which must be at least 16) can be retrieved by querying `glGetIntegerv` with the argument `GL_MAX_VERTEX_ATTRIBS`.

4.4.3 Results

While gathering the data for the graph in Figure 4.3, a maximum batch size of seven targets was used (batch size indicates the number of additions being performed in the vertex shader). The demo was executed multiple times for varying batch sizes. For each run the average frame rate was taken.

These results show that batching the passes reduces the overhead cost of redrawing the geometry during the animation update. Depending on the number of targets used during a given frame, an adequately large batch size should be chosen to reduce this cost.

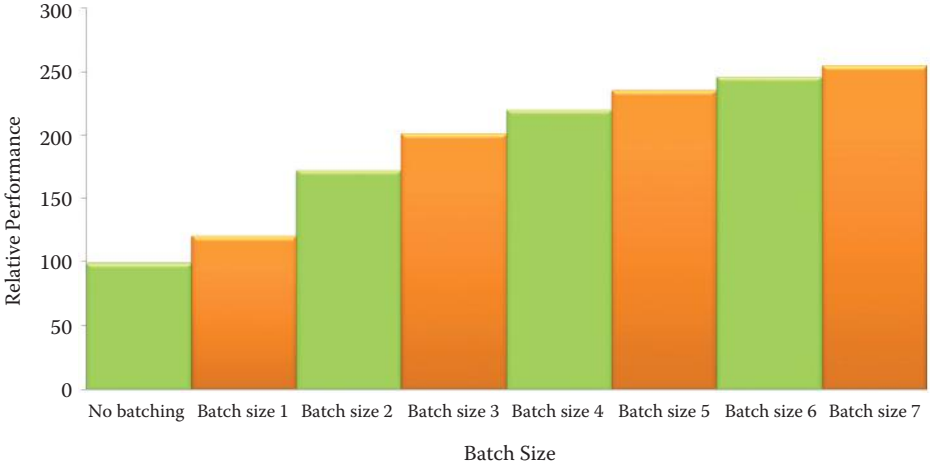

Figure 4.3. Relative performance against batch size when running the demo app.

4.5 Conclusion

This chapter has demonstrated an efficient morph target animation system using transform-feedback. The technique can be computed as a separate pass before rendering, and additional techniques such as skinning can be easily implemented on top of this system.

4.6 Acknowledgements

I would like to thank my colleagues Ken Catterall, Kristof Beets, and Peter Quayle of Imagination Technologies for their consistent support and encouragement.

Bibliography

[Lorach 08] Tristan Lorach. "DirectX 10 Blend Shapes: Breaking the Limits." In *GPU Gems 3*, edited by Hubert Nguyen, pp. 53–67. Upper Saddle River, NJ: Addison-Wesley, 2008.

[Munshi and Leech] Aaftab Munshi and Jon Leech, editors. *OpenGL ES Common Profile Specification*. Khronos Group Inc.

[Senior 09] Andrew Senior. "Facial Animation for Mobile GPUs." In *ShaderX7: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 561–569. Boston: Charles River Media, 2009.