ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2022
D. L. Michels and S. Pirk
(Guest Editors)

Volume 41 (2022), Number 8

CSC 473 Harmonic Coordinates

Kieran Smith

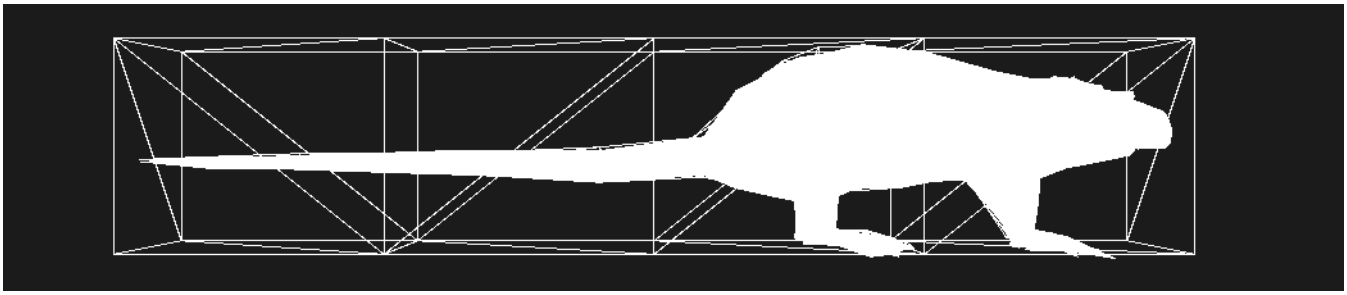

Figure 1: *Reconstructed rat mesh*

Abstract

*In this paper we explore the method of harmonic coordinates described in Harmonic Coordinates for Character Articulation [JMD*07]. Volume deformation algorithms, such as harmonic coordinates, are used by artists for computer animation. We examine implementations of single-threaded, multi-threaded, and GPU-based vertex weight computation to compare time to completion. Our the single-threaded vertex weighting implementation is found to be the most performant. Mean error in the reconstructed vertices is compared for a number of different parameter assignments. Results from insufficient smoothing and excessive quantization error are demonstrated.*

CCS Concepts

• ***Computing methodologies*** → *Collision detection;* • ***Hardware*** → *Sensors and actuators; PCB design and layout;*

1. Introduction

Volume deformation is an essential tool in computer animation. Large animation studios, such as Pixar, require volume deformation algorithms that balance algorithmic performance and visual appeal. Harmonic Coordinates for Character Articulation [JMD*07] is a technique for volume deformation. The majority of its computation may be performed offline, allowing harmonic coordinates to perform mesh deformation in real-time. Results from this method are visually appealing.

Real-time mesh deformation is often achieved using skeletal meshes, which do not preserve visual quality around sharp bends. A mesh is given a skeleton made of bones, and each vertex is assigned weights corresponding to each bone. Skeletal mesh animation is often used in video games because it is trivial to implement and cheap to compute. Such results lack acceptable quality for offline computer animation.

Mean Value Coordinates [Flo03] is a prolific technique for boundary value interpolation. The technique has numerous applications, including 3d texturing, but it is also readily applied to surface deformation. A closed polygonal cage is built to surround a mesh. A function is defined to represent every mesh vertex as a function of each cage vertex. The mesh may be deformed by altering the cage vertex positions. This technique suffers from loss of volume when deforming meshes.

Harmonic Coordinates improves upon Mean Value Coordinates by preserving mesh volume across deformations. Like Mean Value Coordinates, Harmonic Coordinates uses a cage surrounding a mesh in order to deform the mesh's vertices. Harmonic coordinates requires a large amount of processing to compute mesh weights. This computation is only performed once per cage and may be computed offline. Once mesh weights are acquired, it is possible to use harmonic coordinates to deform a mesh in real-time. As such, it is useful for animators who desire immediate visual feedback when animating scenes.

2. Related Work

Mean value coordinates [Flo03] is a widely-used surface deformation method using interpolation across a volume. This technique

has a wide range of applications, including the generation of volumetric textures and mesh deformation. To achieve mesh deformation, a control mesh is constructed to surround a model mesh. Mean value coordinates is used to generate weights for each vertex such that each model vertex is expressed as a function of the positions of the control vertices and a weight vector, as shown in Equation 1. Results of acceptable visual quality may be generated using only a small number of control points. Mean Value Coordinates is able to parameterize a high-resolution mesh in only a few seconds. Deformations may be applied in real-time.

$$\hat{v} = \frac{\sum_j w_j \hat{p}_j}{\sum_j w_j} \tag{1}$$

Skeletal mesh deformation is a mesh deformation technique often used in performance-critical applications, such as video games. Mesh vertices are weighted to a series of bones that compose a skeleton. The position of a vertex is determined by the transformation of each bone it is associated with and weights thereof. Various skinning algorithms are used to generate bone associations and weights. This technique is very fast to compute and is easily rendered a GPU. The hierarchical skeleton used in this technique makes it well-suited to inverse kinematics applications.

The method of harmonic coordinates was first introduced in Harmonic Coordinates [DMS06]. Harmonic Coordinates uses generalized barycentric coordinates [MLBD02] in grid cell weight assignment. The ideas discussed in Harmonic Coordinates are developed further in Harmonic Coordinates for Character Articulation [JMD*07].

3. Overview

Harmonic Coordinates achieves volume deformation by expressing each mesh vertex as a weighted sum of a set of control points. To do this, a closed polygonal mesh referred to as a cage is loaded and placed within a grid of uninitialized cells. Each triangular face of this cage mesh is intersected with the cells of the grid. If a grid cell is in contact with a triangle, it is given weight values corresponding to the barycentric coordinates of its centre with respect to the triangle. After this step is completed, the cells are smoothed until the average change in a cell falls below some value τ. The vertex weights of a given mesh may be extracted from the grid cell containing said mesh vertex.

3.1. Grid Initialization

The grid is initialized by loading a cage mesh from memory. The cage mesh is a closed mesh composed of triangles. A grid of rectangular cells is constructed to contain this cage mesh. This grid is of side length g, having g cells in width, height, and depth. This number is configurable by the user and corresponds to the quantization error caused by the harmonic coordinates parameterization. Each cell in this grid is marked as being either an uninitialized, boundary, exterior, or interior cell. Cells are marked as uninitialized upon initialization.

Our implementation restricts cage specification to quad meshes. This is done for the sake of convenience. It would be trivial to extend our implementation to load cages from arbitrary triangular meshes, though we did not choose to do so.

3.1.1. Mark Boundaries

Boundary cells are identified by intersecting each triangular face in the cage mesh with each cell of the grid. If a cell is intersected by a cage mesh triangle, the cell is marked as a boundary cell. Each boundary cell holds a vector of weight values with size equal to the number of vertices in the cage mesh. Each entry in this vertex is initialized to zero. Should a boundary cell intersect with a triangle with constituent vertex i of the cage, then the cell's weight at index i is equal to the barycentric coordinate the cell's centre corresponding to vertex i in its triangle. An algorithm to compute the initial values of a cell is shown in Algorithm 1.

A top-down approach, iterating over every triangle to examine the cells it intersects, was chosen for our implementation. Our implementation tests for intersection of each triangle for every cell in the grid. It is possible to examine only the cells intersecting the triangle's axis-aligned bounding box. Doing so will vastly reduce the computation required to identify collisions between the cells of the grid and small triangles.

A bottom-up approach, iterating over every cell to test for intersection with every triangle, is more suited to parallelization. The marking of boundary cells is a bottleneck in our implementation. Parallelization of boundary marking may be examined in future work.

Algorithm 1 Boundary cell marking algorithm

```
for triangle in cage do
  if intersects(cell, triangle) then
    c ← centre(cell)
    b ← barycentric(c, triangle)
    i ← 0
    for j in triangle do
      w_j ← b_i
      i ← i + 1
    end for
  end if
end for
```

3.1.2. Mark Exterior

Following the boundary marking step, a flood-fill algorithm is run from the cells on the exterior of the grid. Any uninitialized cell encountered is marked as an exterior cell. Our implementation makes use of breadth-first search for this step in the computation. Other traversal algorithms, such as depth-first search, would be functionally equivalent. Pseudocode for exterior marking may be found in Algorithm 2.

Our flood-fill is started from the zeromost cell. If the zeromost cell is cut off from other exterior cells, then the exterior cells that are cut off from the zeromost cell will be marked as interior cells. This does not result in visual error, but does result in additional smoothing computation. A potential solution involves marking all outermost cells of the grid as start points for our flood-fill algorithm, though we did not choose to do so.

Algorithm 2 Exterior cell marking algorithm

```
q ← []
push([0,0,0])
while q not empty do
  c ← pop()
  if s_c = uninitialized then
    s_c ← exterior
    for neighbouring cell do
      push(neighbouring cell)
    end for
  end if
end while
```

3.1.3. Mark Interior

After the boundary and exterior cells have been marked, any uninitialized cell is then marked as being an interior cell. Our implementation stores an array representing the state of each cell (uninitialized, boundary, exterior, or interior) and another array for the cell weights. Because only boundary and interior cells are assigned weights, it is unnecessary to store weight data for exterior cells. Each cell state may be represented with as few as four bits. Reduced memory usage is beneficial for GPU-based smoothing, as it reduces the amount of data transferred and allows for more efficient cache utilization.

3.2. Smoothing

Smoothing is performed by setting each interior cell's weights to be the average of its neighbours. Cell weight data is copied and stored within a buffer, so the additional memory required for smoothing is that of the memory required to store the grid's weight data. This is continued until the average change in a cell's weight vector, as shown in Equation 2, falls below a user-configured threshold τ. In our examples, τ is usually within the range of 10^{-4} to 10^{-5}. An algorithm to compute the next iteration's cell's weights is shown in Algorithm 3.

$$delta = \sqrt{\sum_{i=0}^{|v|} |w_i^n - w_i^{n-1}|} \tag{2}$$

Algorithm 3 Smoothing algorithm

```
for interior cell do
  w_accum ← 0
  count ← 0
  for neighbouring cell do
    w_new ← w_new + w_neighbour
    count ← count + 1
  end for
  if count ≥ 1 then
    w_new ← w_accum / count
  end if
end for
```

A large value of τ will result in a large reconstruction error.

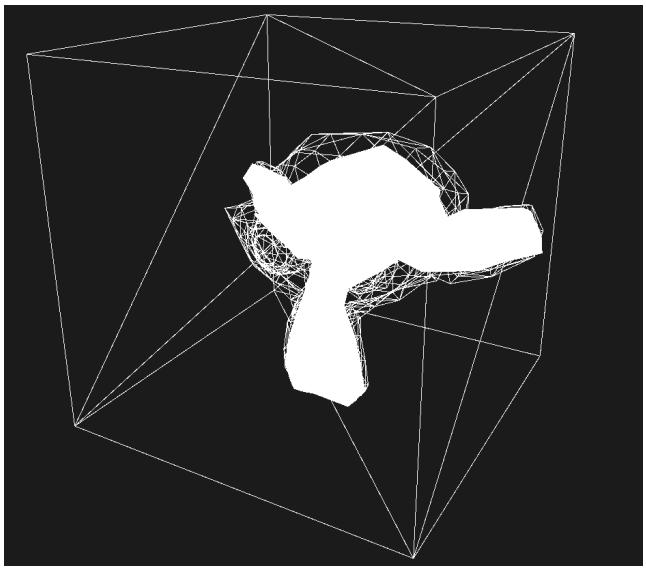

Figure 2: *Suzanne reconstructed with* $g = 32$ *and* $\tau = 10^{-3}$

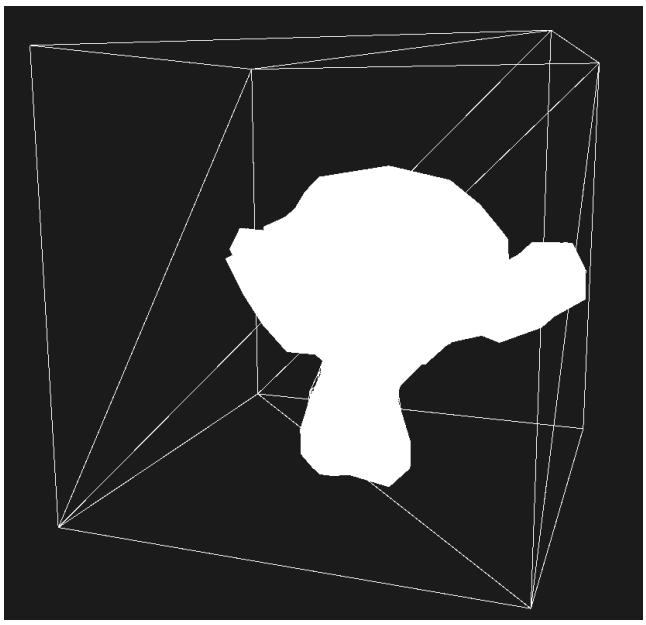

Figure 3: *Suzanne reconstructed with* $g = 32$ *and* $\tau = 10^{-5}$

This is visually demonstrated in Figure 2, showing a reconstruction based on a grid smoothed to a τ value of 10^{-3}. The base mesh is rendered as a wireframe, and the reconstructed mesh is rendered as a solid. We see that the reconstructed mesh is significantly smaller than the base mesh. This is in contrast to Figure 3, showing the same mesh with the same g value but smoothed to a τ value of 10^{-5}. This rendering so closely approximates the original that the wireframe is barely visible.

3.2.1. Parallel Smoothing

Each iteration of cell weightings is dependent only on the iteration before it. This allows for a large degree of parallelization. Our implementation features options for smoothing using multithreading on the CPU and compute shaders on the GPU.

When smoothing in parallel on the CPU, our implementation spawns a thread for each available core on the CPU. Each of these threads consumes work from a queue. The queue is loaded with the grid position of each interior cell. Smoothed weight data is written to the output buffer as in the single-threaded smoothing algorithm. In theory, the memory used in multi-threaded smoothing is approximately equal to the memory used by single-threaded smoothing.

GPU-based smoothing in our implementation makes use of a compute shader. This shader reads from a cell state buffer and cell data buffer. Outputs are written to a duplicate cell data output buffer. After each dispatch of the smoothing shader, the contents of the cell data output buffer are written to the cell data buffer. Average change in weight values is computed every c iterations, where c is a non-zero value specified by the user. Once the average change in weight values is computed, it is copied to main memory along with the contents of the cell data buffer. At this point the CPU reads the reported value and compares it to τ. If the reported value is greater than τ, another batch of c smoothing iterations is dispatched. Otherwise, the contents of the cell data buffer are stored within main memory and we are ready to move on to the mesh weighting step.

Allowing the user to specify c improves performance by reducing the number of memory transfers between the CPU and GPU. The drawback to this approach is that more smoothing is performed than is necessary. If c is 10 and average change in weight vector is less than τ after iteration 0, the program will still perform 9 more iterations before terminating computation.

It is possible to remove the cell data memory copy after each smoothing iteration by enabling read and write access for both the cell data input buffer and output buffer. In order to do so, however, the shader would need to read and write to another tracking buffer to decide which buffer to use for input and which to use for output. Furthermore, the contents of this tracking buffer would need to be accessed by the CPU when cell weight data is to be transferred back into main memory. We view this workaround to be too complicated for an uncertain improvement in performance.

CPU-GPU memory transfer is a known bottleneck in graphics applications. Our implementation transfers the entire contents of the GPU's cell data buffer to the CPU after every c smoothing passes. It is possible to wait until a sufficiently low delta is reached and perform only one of these transfers. Doing so would introduce one additional work submission to the GPU.

3.3. Mesh Weighting

Each mesh vertex is given the weights of the grid cell it occupies. This is extremely fast to compute, but introduces further quantization error. Our implementation introduces quantization error in this step, when fetching the weights for each vertex from the grid, and also when applying weight from the cage onto the grid. An example demonstrating the consequences extreme quantization error may be seen in Figure 4.

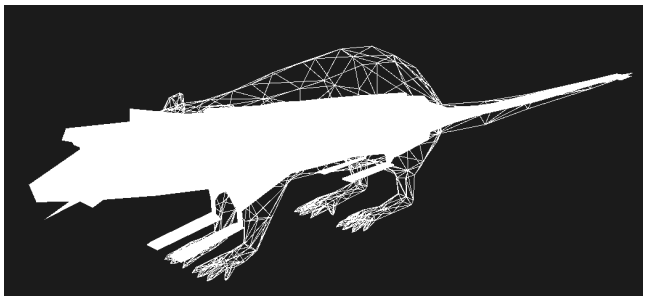

Figure 4: *Visualization of quantization error with* $g = 16$ *and* $\tau = 10^{-5}$ *with base mesh is rendered as wireframe*

Figure 5: *Reconstructed rat mesh with* $g = 32$ *and* $\tau = 10^{-5}$

4. Evaluation

Our implementation was evaluated visually and found to be adequate. Figure 5 show a reconstruction of a rat mesh. Cage-based deformation is possible in real-time. Figure 6 shows deformation of a reconstructed rat mesh through alteration the cage vertex positions.

All data included in this section was generated on EndeavourOS Linux x86_64 with CPU Intel i7-7700HQ, GPU NVIDIA GeForce GTX 1050 Mobile, and 8GB of RAM. Our implementation may be found online and was compiled with rustc 1.77.2.

4.1. Performance

Comparison of time to completion may be found in figures 8, 9, 10, and 11. A graph of delta over time is shown in Figure 7. Delta appears to decrease logarithmically. None of our test cases required more than four minutes to compute. Future work might measure the computational time required to create parameterizations with greater g values and lesser τ values. Interestingly, both threaded and GPU-based smoothing were slower to reach completion than the single-threaded implementation.

The threaded smoothing algorithm requires additional overhead to allow threads to pull work from the queue. Our implementation queues work for each cell in the grid. An implementation which queues groups of cells for processing, rather than individual cells, would avoid this overhead. Furthermore, our implementation failed to make use of proper synchronization structures in the Rust programming language. As a result, each cell must acquire a mutex lock before writing is data into the result buffer. There is significant

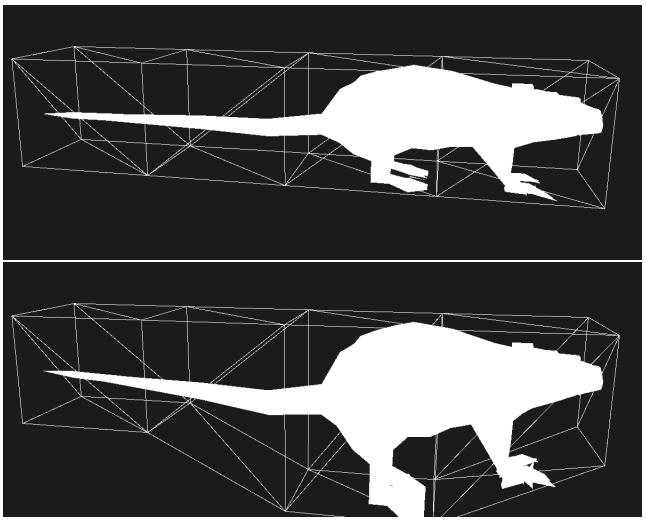

Figure 6: *Rat deformation*

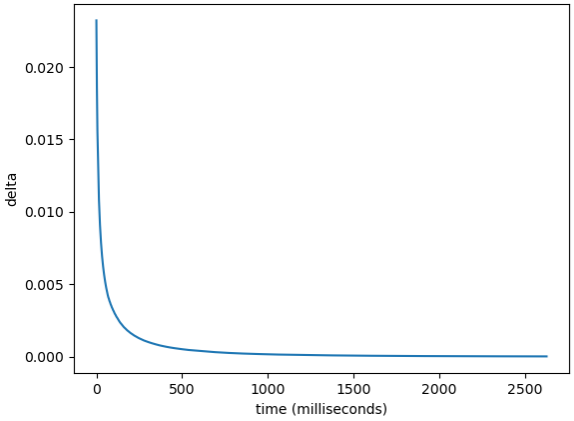

Figure 7: *Delta over time with* $g = 32$ *and* $\tau = 10^{-5}$

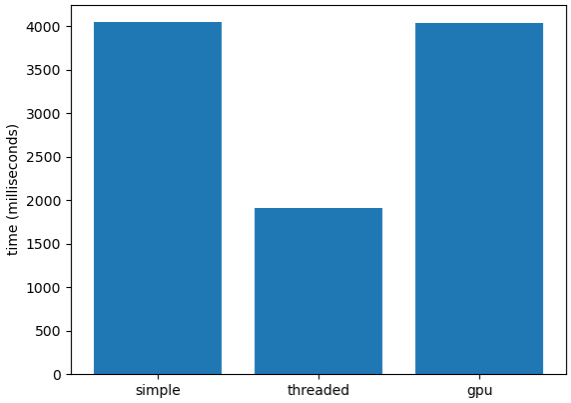

Figure 8: *Time to completion with* $g = 32$ *and* $\tau = 10^{-4}$

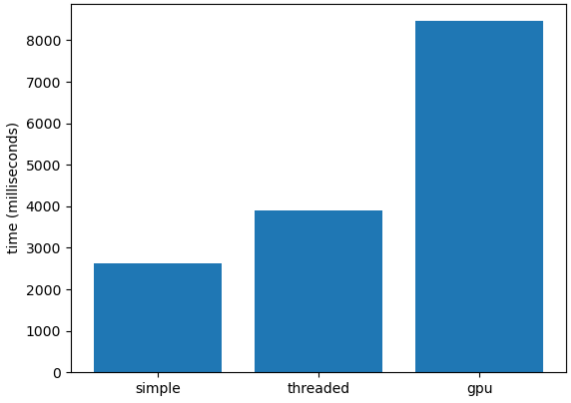

Figure 9: *Time to completion with* $g = 32$ *and* $\tau = 10^{-5}$

potential for improvement in the implementation of the threaded smoothing algorithm.

The GPU-based smoothing algorithm transfers the contents of the smoothed data buffer to the CPU after every 10 smoothing steps. The transfer of memory between the CPU and GPU memories is a large bottleneck in the algorithm. Increasing the number of smoothing steps performed per iteration will improve the algorithm's performance. The increased number of smoothing steps per iteration will also result in excess work, iterations performed after the smoothness threshold has been reached.

Our single-threaded smoothing may exhibit increased performance due to the benefits of greater cache-locality. Because it is operating upon individual segments of weight data that are stored near to other segments of weight data in main memory, it is able to hold more relevant data in cache memory.

4.2. Reconstruction Error

Mean reconstruction error was computed using equation 3. A graph of measured across various grid sizes and tau values is shown in figure 12. Mean error is expected to decrease with lower values of tau. This was shown to be true with a grid size of 64, but the mean error value remained similar with grid size 32.

$$error = \sum_{i=0}^{n} \frac{|reconstructed_i - original_i|}{n} \tag{3}$$

Figure 12 shows a large mean error that for grid size 64 and tau 10^{-4}. This discrepancy is a result of larger grid size values requiring a lower value of tau. Increasing the grid size increases the number of interior cells. The rate of smoothing, however, remains constant. These two factors can result in artificially low average

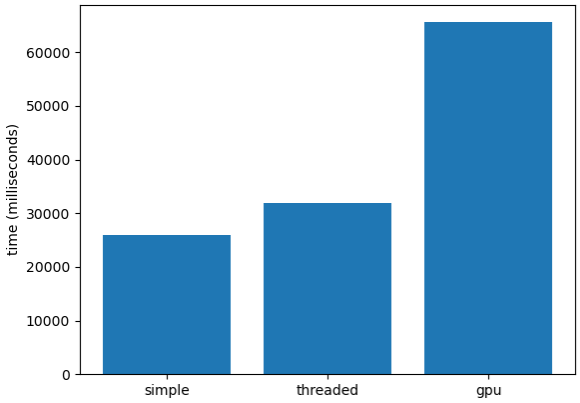

Figure 10: *Time to completion with* $g = 64$ *and* $\tau = 10^{-4}$

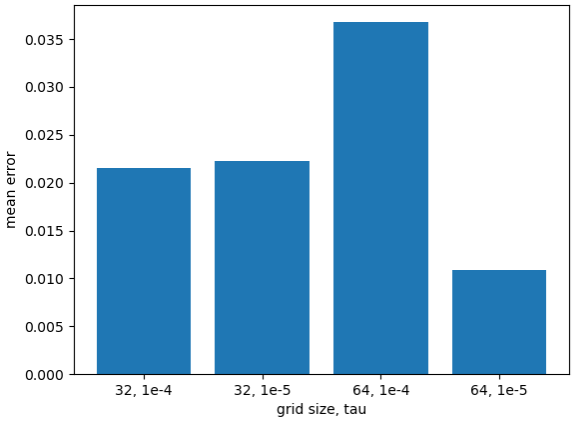

Figure 12: *Mean error with various parameters*

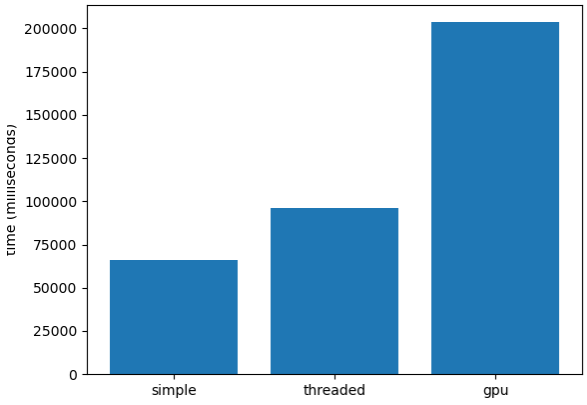

Figure 11: *Time to completion with* $g = 64$ *and* $\tau = 10^{-5}$

change in weight values, resulting in premature termination and low-quality reconstruction.

5. Conclusion

Our implementation of harmonic coordinates is evaluated visually, in terms of performance, and in terms of accuracy. Visual results created by the algorithm are appealing. The algorithm demonstrates acceptable performance characteristics. Our exploration of parallelization for grid smoothing is found to have a negative impact on time to completion. Meshes are reconstructed accurately given adequate tuning of the input parameters g and τ. Low values of g result in excessive quantization error. High values of τ result in insufficient smoothing. In general, higher values of g and lower values of τ result in a lower mean error in reconstructed vertex positions.

Challenges in the development of this project centred around the debugging of shaders. Our smoothing shader and display shaders both required tuning. RanderDoc was found to be an extremely helpful tool for shader debugging. Program structure in our implementation lacks sufficient planning. Confusion as a result of this lack of clarity led to inefficiency in parallel smoothing.

Certain values of g and τ are shown to produce visually unappealing and inaccurate results. GPU-accelerated grid smoothing is limited by the GPU's maximum allowed buffer size. Our implementation does not make use of a depth buffer. The use of a depth buffer would allow for more visually interpretable results.

Evaluation may be explored using a more efficient implementation. Much of our implementation is naive and unoptimized. Additional evaluation may be conducted with greater values of g and lesser values of τ. Several extensions described in Harmonic Coordinates for Character Articulation [JMD*07] are unimplemented in this project. Future work may implement hierarchical grid data structure, fine and course grid data, interior control structure, and dynamic binding as described in Harmonic Coordinates for Character Articulation [JMD*07].

References

[DMS06] DEROSE T., MEYER M., STUDIOS P.: Harmonic coordinates. *Pixar Technical Memo* (01 2006).

[Flo03] FLOATER M.: Mean value coordinates. *Computer Aided Geometric Design 20* (03 2003), 19–27. doi:10.1016/S0167-8396(03)00002-5.

[JMD*07] JOSHI P., MEYER M., DEROSE T., GREEN B., SANOCKI T., STUDIOS P.: Harmonic coordinates for character articulation. *Pixar Technical Memo* (07 2007). doi:10.1145/1239451.1239522.

[MLBD02] MEYER M., LEE H., BARR A., DESBRUN M.: Generalized barycentric coordinates on irregular polygons. *Journal of Graphics Tools 7* (07 2002). doi:10.1080/10867651.2002.10487551.