Efficient Triangle Reordering for Improved Vertex Cache Utilisation in Realtime Rendering

Martin Storsjö

Master of Science Thesis
Supervisor: Tomi Aarnio, Tero Nordström, Nokia
Examiner: Jan Westerholm
Department of Information Technologies
Faculty of Technology
Åbo Akademi University

Abstract

Storsjö, Martin: Efficient Triangle Reordering for Improved Vertex Cache Utilisation in Realtime Rendering

Master's Thesis, Department of Information Technologies, Faculty of Technology, Åbo Akademi University, 2008. 100 pages, 18 figures, 14 tables.

Keywords: Vertex cache, embedded, mobile, overdraw, M3G, computer graphics

In this thesis, realtime rendering optimisations based on the reordering of data and changing of data format are evaluated. The optimisations include improved vertex cache utilisation, overdraw reduction, improved vertex data read patterns and conversion of inefficient triangle strips into triangle lists. The improvements are evaluated by integrating them into existing high-level 3D graphics frameworks for embedded environments such as M3G. In particular, fast algorithms for reordering triangles in order to improve the utilisation of vertex caches are studied. The memory usage and run time performance of the algorithms are improved by making small compromises in the end result.

Acknowledgements

When describing this thesis, there is one name which should be mentioned above all other, Tomi Aarnio. Tomi has helped immensely while I have been implementing, testing and evaluating these optimisations. Without his in-depth knowledge and experience, this thesis would never have turned out the way it did. In fact, I would not even have known about this subject if he would not have pointed out this as an area I could research.

Additionally, I would also like to thank Tero Nordström for all the support and help during the whole process, Jan Westerholm for all the helpful directions and comments and Mia Westerlund and Johan Schöring for proofreading all this. I would also like to thank Nokia for sponsoring this research and the Stanford 3D Scanning Repository, Kishonti Informatics LP and HI Corporation for the test models I have been using.

Åbo, May 13, 2008

Martin Storsjö

Contents

List of Figures

List of Tables

Listings

Abbreviations

ACMR	Average Cache Miss Ratio
API	Application Programming Interface
FIFO	First In, First Out
IPT	Indices Per Triangle
LRU	Least Recently Used
M3G	Java Specification Request 184: Mobile 3D Graphics API for Java ME

Chapter 1

Introduction

1.1 Motivation

In recent years, mobile phones have evolved from simple communication appliances to full-featured, general-purpose mobile computers. They are acquiring more features that were earlier only available on desktop computers in rapid succession and are merging features from many devices into one single device.

The graphics capabilities of mobile phones have grown in a similar way. Earlier, the screens were monochrome and had low resolution, capable of displaying a few rows of text and simple icons. Lately they have grown both in resolution and in size and become able to display colour with high precision.

As mobile phones become more capable, their software needs to grow to utilise the capabilities. 3D graphics will play a more important role, both in games and in the user interface in general. On desktop computers, 3D graphics is used in almost all commercial computer games at the moment, and mobile games will probably develop in the same direction. The user interfaces will use 3D graphics to become more user-friendly, intuitive and visually attractive.

Even though the mobile environments grow to become more capable, they will always have more limiting factors than the desktop environments. The processing power and the available memory are usually much more limited than on desktop platforms. Battery capacity limits the energy consumption, which indirectly limits how powerful chips can be used. Optimisations at all levels will still be necessary, due to the inherent limitations in the mobile environments.

One way of overcoming some of these limitations within realtime rendering of 3D

graphics is by effectively utilising the vertex transform caches, as described in this thesis.

1.2 Thesis structure

Initially, the concepts of realtime 3D graphics needed by the rest of the thesis are described in chapter 2, followed by a presentation of a few areas where the rendering overhead can be reduced in chapter 3. The problem the thesis aims at solving is formulated together with criteria for evaluating the solution in chapter 4.

The algorithms analysed in the thesis, two triangle reordering algorithms for vertex cache utilisation and one for overdraw reduction, are introduced in chapter 5, followed by an analysis of their initial memory usage in chapter 6. Then the environments used for testing are described, together with the content used for the tests, in chapter 7. The specific properties of one algorithm is analysed in detail in chapter 8. Improvements to the algorithms are presented, including analysis of the compromises, in chapter 9.

The two algorithms improving the vertex cache utilisation are compared in chapter 10. The rendering time improvements offered by the optimisations are presented and discussed in chapter 11. Integration into an existing scene graph framework is discussed and benchmarked in chapter 12, followed by reiteration of the results of the thesis and pointers on potential future work in chapter 13.

Chapter 2

Realtime rendering

The process of creating a viewable image from a geometrical description of the content is called rendering. Realtime rendering is the case where every image is rendered immediately when it is to be displayed, with a short rendering time for each frame. This allows interactive animation for example in games, since the source data for each rendering can be changed between each rendered frame based on user input.

The routines for doing rendering can be generalised into reusable libraries, to be used by many applications and games. Libraries for interfacing with graphics hardware are usually provided by the hardware vendor. By standardising application programming interfaces (API) for these kinds of libraries, applications need not be tied to any specific implementation but can be portable across both software and hardware based renderers.

2.1 3D graphics APIs

OpenGL is a cross-platform low-level graphics API, initially introduced by Silicon Graphics in 1992. Implementations of this API exist for most dominant desktop platforms (Microsoft Windows, Linux, Apple Mac OS X) and many of the less dominant platforms. [SAF+06]

OpenGL ES is mainly a subset of OpenGL, intended for implementation on embedded systems. It is a simplified specification, where little-used parts of the OpenGL standard have been removed, to make implementations smaller without radically compromising features needed in practice. OpenGL ES also includes some additions needed specifically on embedded platforms. Due to the

large similarities between OpenGL and OpenGL ES, large parts of applications are source-level compatible. Implementations of OpenGL ES are available for many mobile platforms (Symbian, Microsoft Windows Mobile, Apple iPhone and BREW among others). [BM07]

M3G (Mobile 3D Graphics) is a slightly higher level API for 3D graphics for Java Micro Edition (Java ME). It is designed to be a simple but high performance API, efficiently implementable on top of OpenGL ES. It provides scene graph based and immediate mode rendering. To enable efficient and potentially hardware accelerated implementations, all mesh data is encapsulated into opaque objects where the data representation is hidden from the user of the API. Encapsulating the data representation allows for further optimisations of the mesh data. [AEHR05]

OpenGL ES and M3G are currently the two main cross-platform 3D graphics APIs for mobile platforms.

2.2 Triangle meshes

2.2.1 Mesh geometry

In most common implementations of realtime 3D graphics, the geometry is constructed from a set of vertices and a mesh of polygons defined using the vertices. Each vertex has a number of attributes: position, normal vector, colour, texture coordinates etc. For simplicity, many implementations limit the polygons to triangles, since all polygons can be constructed from one or more triangles.

The number of triangles t can at most be twice the number of vertices v, as explained by Bar-Yehuda and Gotsman [BYG96]. Therefore, many calculations can be simplified using the approximation $t \approx 2v$.

2.2.2 Triangle data formats

The mesh data can be stored in several different formats. The formats described below are supported by OpenGL and OpenGL ES. The simplest format is one where all the vertex data is specified explicitly for all three vertices in each triangle. Instead of explicitly specifying all the vertex data every time a certain vertex is used, the vertex data can be stored in one separate buffer or a separate buffer for each vertex attribute, and thus triangles are specified only using vertex

indices. This approach has a multitude of advantages, among others it requires less memory to store the mesh, as explained by Hoppe [Hop99]. Additionally, if updating the vertex attributes (e.g., when animating a mesh), the data for each vertex only needs to be updated at one single place in the vertex data buffer. These formats are illustrated in figure 2.1.

The order in which the vertices within a triangle are specified does matter. By keeping a consistent ordering, it is possible to differentiate the front side and back side of triangles by specifying that viewed from their front side triangles are defined in, for instance, counter-clockwise order.

A potentially more efficient format is the so called triangle strip format. In that format, a new vertex (either complete set of vertex data or a vertex index) specifies a new triangle, consisting of the two previous vertices and the newly specified one. That is, every triplet of consecutive vertices in the strip generates one triangle. If two of the vertices in a triplet are the same, the triangle is considered to be degenerate and will not be rendered.

As long as an edge between two vertices is shared by at most two triangles, there is only one single candidate for the next triangle in a strip when constructing the strips. When the strip cannot be continued further, a completely new strip must be created or degenerate triangles must be inserted. Figure 2.1 contains an example where a mesh is constructed from two separate strips and another example where the two separate strips have been merged into one single strip containing degenerate triangles. All triplets up to (5, 3, 6) generate normal triangles, but the triplets (3, 6, 6), (6, 6, 4), (6, 4, 4) and (4, 4, 7) contain multiple occurrences of a single vertex, making them degenerate and preventing them from being rendered.

The winding order is defined to be alternating. Every other triplet (starting with the first one) generates a triangle with indices in the same order as in the strip, the rest of the triplets generate triangles with opposite order. Therefore, the strips in the example generate the triangles (1, 4, 2), (2, 4, 5), (2, 5, 3) and so on, in correct and coherent winding order. When determining the winding order, degenerate triangles are included, which allow to determine the winding order of a triplet directly based on the offset from the start of the triangle strip.

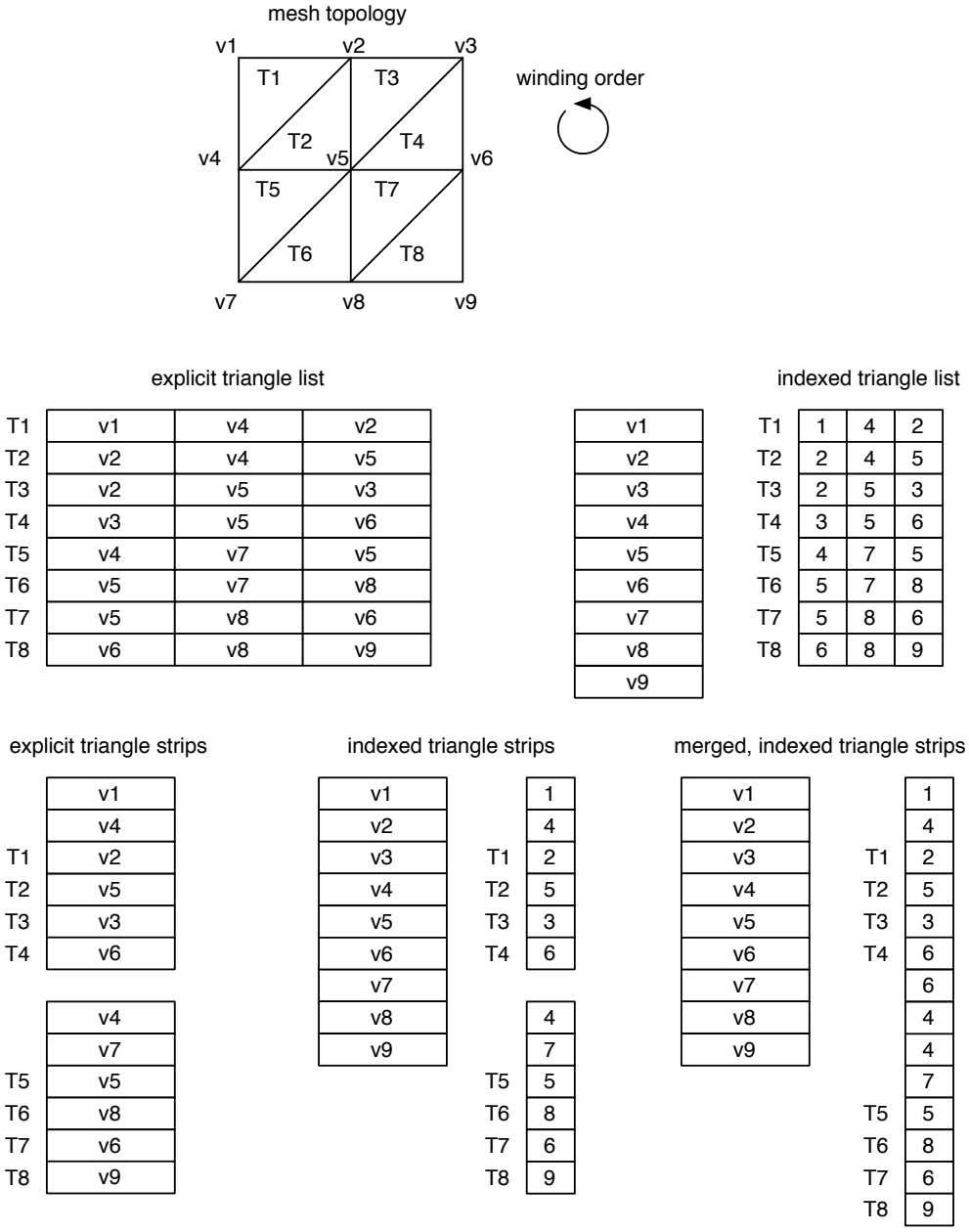

Figure 2.1: Illustration of mesh data formats, based on illustrations by Hoppe [Hop99].

2.2.3 Mesh drawing

When rendering the meshes, one triangle at a time is rasterised and drawn to the target (either the screen or another buffer), updating the colours of the pixels covered by the triangle.

A method named depth buffering, z-buffering or depth testing is often used at this stage of the rendering process. The main concept of the method is keeping track of the depth coordinate for every rendered pixel while the 3-dimensional scene is rendered. When a new triangle is rasterised, the depth value for each pixel is compared to the previous value. Only if the new triangle is closer to the virtual camera than the current content of that pixel, the pixel colour and depth value are updated, otherwise the previous colour and depth values are kept unmodified.

Depth buffering allows the different meshes of a scene to be rendered in any order, creating the same correct output in all cases. It also allows the triangles within a mesh to be drawn in any order.

This freedom is not available in all cases, though. When drawing geometry with blending, where the new colour drawn depends on the previous colour, the drawing order cannot be varied freely without affecting the output result. However, if depth buffering is disabled and the geometry is drawn with additive blending, the drawing order can be varied freely.

Chapter 3

Ordering and format optimisations

Below, a few optimisations based on reordering or converting triangle or vertex data are presented.

3.1 Caching of transformed vertices

3.1.1 Vertex transforming overhead

During rendering, the vertices are transformed from their coordinates within the object definition to coordinates in the world coordinate system. The 3D world coordinates are projected into 2D coordinates in a plane, according to the position, rotation and projection type of the virtual camera. The vertex normal vectors are transformed according to the object rotation and used for the calculation of lighting coefficients. These transformations are described in detail in, for instance, OpenGL® Programming Guide [SWND05].

The transformations are potentially heavy operations. They are usually implemented with floating point arithmetics, even though the embedded platforms might lack a hardware floating point unit. For optimal performance, as few transformations as possible should be made.

Traditionally, rendering has been designed in a pipeline-like fashion, where single geometry primitives (e.g., triangles) are sent down the pipeline one at a time. In this setup, every vertex is transformed every time it is used. Since most vertices are used by more than one triangle, each vertex might be transformed more than once, which is in principle redundant work.

One potential solution is to transform all vertices before rasterising the triangles, which guarantees that no vertex is transformed more than once. However, this is not done in practice, for a number of reasons. The intermediate result from the transformations need to be stored, potentially requiring large buffers.

The vertex and index buffers are decoupled when using indexed triangle lists or strips, and the same vertex buffers can be used by many different index buffers which each might use a subset of the vertices. Therefore, all vertices may not be used. In some level-of-detail setups, there can be several versions of the same mesh with different levels of geometry detail constructed by larger or smaller subsets of the vertices.

3.1.2 Previous work

In 1996, Bar-Yehuda and Gotsman showed that every triangle mesh with n vertices can be reordered so that rendering needs only an intermediate buffer of size $O(\sqrt{n})$ for the transformed vertex data, without transforming any vertex more than once [BYG96]. Therefore, it is possible to avoid all vertex transforming overhead with a much smaller buffer than one for transforming all vertices before rasterisation.

Bar-Yehuda and Gotsman also presented algorithms for creating such triangle orderings, in a format where commands for managing the intermediate buffer for transformed vertex data are explicitly intermixed with the instructions for constructing triangles from vertices.

In 1999, Hoppe presented the idea that the intermediate buffer could be implemented as a transparent FIFO (first in, first out) cache [Hop99]. Whenever the renderer needs a vertex, it checks whether it already is transformed and resides in the cache. If it is there, the data is reused, otherwise the vertex is transformed and the result is added to the cache, overwriting the oldest entry. In that way, the same triangle data works on any implementation, regardless of the size or even presence of the intermediate buffer. This also made the concept easily implementable without changing the existing graphics APIs.

3.1.3 Measuring vertex caching

When the renderer has a transparent vertex cache, any triangle order works, but with varying performance. The vertex transforming overhead can be measured

by calculating the average cache miss ratio (ACMR), that is, the number of cache misses divided by the number of triangles rendered. For each cache miss, a vertex is transformed and added to the cache.

In the worst case, no vertices are ever found in the cache and all three vertices in all triangles need to be transformed, giving an ACMR of 3. The best case is where each vertex is transformed at most once. Since the number of triangles is less than twice the number of vertices, the ACMR cannot be less than 0.5.

3.1.4 Triangle reordering algorithms

A number of algorithms have been proposed for generating drawing orders in order to utilise vertex caches as well as possible. Much research has been done on generating efficient triangle strips, among others by van Kaick et al. [vKdSP04]. Other algorithms generate orders not constrained to the strip format, some optimising for specific cache sizes, e.g. K-Cache-Reorder by Lin and Yu [LY06], and others aiming at giving good cache utilisation regardless of the cache size, e.g. an algorithm by Bogomjakov and Gotsman [BG02].

Many algorithms have a high run time complexity, which is not a big problem if they run only when the content is generated, but makes them unsuitable for usage in the target environment.

In 2006, Forsyth introduced an fast algorithm with a linear run time for generating universally efficient rendering orders [For06]. Sander, Nehab and Barczak presented another fast, linear time algorithm in 2007 named tipsify, optimising for specific cache sizes [SNB07]. These two fast algorithms might not achieve as good ACMR as the best algorithms, but it is feasible to implement them in the target environments, too, not only in the content generation stage. These two algorithms are the ones analysed, evaluated and improved upon in the following chapters.

The importance of and potential gains by reordering the triangles can be illustrated by the results in, for example, table 11.1, showing an over 60% increase in frame rate on the software implementations on mobile phones and much over 100% increase on laptop chipsets.

3.2 Vertex reordering

When rasterising triangles, transformed data for the vertices is needed. If the transformed vertex data is cached, it is reused, otherwise the original data must be fetched and transformed. For optimal performance, data should be read from the original buffers as sequentially as possible. The optimal read pattern of course depends on the actual memory implementation and its cache structure, but usually larger consecutive blocks are fetched into the cache at once.

The read pattern can be changed by reordering the actual vertex data in the vertex buffers (giving each vertex a new index number) and updating the index buffers accordingly. An ordering giving a sequential read pattern can be generated by creating a mapping from old vertex indices to new ones. The index buffers, containing vertex indices in the orders they will be used, are traversed. If no mapping exists for the old vertex index, it is mapped to the next unused new vertex index. The vertex data is reordered using this mapping, and the indices in the index buffers are updated.

Note that this kind of reordering does not change the topology or drawing order of the mesh in any way. It only changes the storage layout of the vertices and gives them new names, i.e., indices.

With this kind of ordering, the vertex data to use is either located directly after the previous read, or is a vertex which has already been used. If it is a reused vertex, chances are that it is still in the vertex transform cache, especially if the triangle ordering is optimised. Only the vertices which are reused but are not found in the vertex transform cache must be refetched from the original buffers in a non-sequential order.

3.3 Overdraw reduction

When drawing any nontrivial scene, many pixels will be rasterised as part of several different triangles. If the so called painter's algorithm it used, the whole scene is drawn from the back to the front. In this scenario, the pixels covered by more than one triangle are drawn and updated many times. If depth buffering is used instead, the triangles can be drawn in any order. If the depth test indicates that the pixel shouldn't be updated, the colour of the pixel need not be computed at all.

If the computing of the colour is a heavy operation (e.g., using expensive texture filtering or a heavy fragment shader), much unnecessary work in the rendering process can be avoided by making sure every pixel gets redrawn as few times as possible, that is, drawing the visible triangles first and the ones which are more occluded later.

3.4 Triangle data formats

The triangle strip format seems like an efficient format at a first glance. If the whole mesh would consist of one single strip, it would have an ACMR of at most 1.0, disregarding the negligible effect of the initial two transformed vertices which do not generate any triangles, since every new transformed vertex generates one triangle. Additionally, the single new vertex for each triangle may already be cached, lowering the ACMR further.

In practice, though, one single triangle strip is usually not enough to traverse the whole mesh, but many shorter triangle strips are needed. To draw a mesh consisting of many strips, each strip could either be drawn individually, or the strips could be spliced into one long strip. The overhead of starting drawing is usually high, making splicing the strips the better alternative.

When triangle strips are spliced, extra indices must be inserted in-between the two strips, to make sure no extra triangles are generated from the index triplets in the joins between strips. If the first strip ends in (. . . , 7, 8, 9), and the following strip starts with (20, 21, 22, . . .), the splice area would be (. . . , 7, 8, 9, 9, 20, 20, 21, 22, . . .). The original triplet (7, 8, 9) still generates a triangle, but the triplets (8, 9, 9), (9, 9, 20), (9, 20, 20) and (20, 20, 21) are degenerate since one index appears more than once and thus are discarded and generate no triangles. That is, the last index from the first strip and the first index from the following strip are duplicated.

When joining triangle strips, the winding order must also be considered. Therefore, if the prepended strip is of odd length, the winding order of the following strip will be reversed. To avoid this a third extra index can be added in the splice, making the prepended strip to be of even length.

These factors affect the efficiency of the triangle strips. One way of measuring the efficiency is to divide the total number of indices in the joined strip with the number of real, non-degenerate triangles in the strip, giving the IPT (indices per triangle) metric.

The IPT metric is similar, but not identical to the ACMR metric. Efficient triangle strips have IPT values slightly over 1 (values below 1 are not possible). A single triangle strip with no degenerate triangles will have an IPT converging to 1 when the length of the strip grows. The other extreme case is when joining triangle strips where each strip consists of one single triangle. Since a single triangle is a strip of odd length, three padding indices are needed between each strip. The joined strip therefore has 6 indices per triangle (except for one triangle).

For triangle lists the IPT value is fixed at 3, since there are no degenerate triangles and every triangle is explicitly defined with the indices of all three vertices.

When reordering triangles generally, the new order is output as a triangle list, since that is the only format where any arbitrary order can be specified easily. Therefore, a conversion from a well-conditioned triangle strip with an IPT value close to 1 will yield a triangle list with a much higher IPT value. The memory needed and memory bandwidth used for the index buffer is directly proportional to the IPT value. Even though the reordering yields a drawing order with a lower ACMR, the larger IPT might outweigh the ACMR improvement in some cases.

On the other hand, a bad triangle strip representation may have an IPT value over 3. In that case, simply converting the triangle strip to a triangle list saves memory usage and bandwidth, even without doing any reordering at all. Triangle strips can trivially be converted to triangle lists, while the reverse conversion is more complex if efficient strips are desired.

Chapter 4

Problem statement and evaluation criteria

4.1 Problem statement

In mobile, other embedded or very resource limited environments, the rendering must be highly optimised. Large compromises regarding the visual quality usually have to be made in order to get acceptable performance. In practice, the drawing orders of most existing 3D models are more or less suboptimal, and by drawing them in a more efficient order, performance can be improved without sacrificing the visual quality at all.

In this thesis we focus on the following four tasks:

1. We evaluate two existing fast vertex cache optimisation algorithms, tipsify and Forsyth's algorithm, for usability in resource limited environments.
2. If possible, we improve their run time performance and memory usage.
3. We evaluate whether this could be enabled universally in the underlying frameworks (e.g., an M3G library), to benefit all users without modifying the existing applications and content.
4. Other closely related optimisations, vertex reordering and overdraw reduction are also evaluated.

4.2 Evaluation criteria

The optimisations must be so fast that they can be implemented in a mobile environment without significantly slowing down the loading process, and must not require much more memory than the actual content itself.

The optimisations must improve the rendering performance on average and must not degrade performance remarkably in any single case. They must be integratable into the existing frameworks, working automatically and transparently.

Chapter 5

Triangle reordering algorithms

In the following sections, two algorithms for reordering triangles for improved vertex cache utilisation, called tipsify and Forsyth's algorithm, and one for reordering for reducing overdraw, called view independent overdraw reduction, are presented.

5.1 The tipsify algorithm

The tipsify algorithm was presented by Sander, Nehab and Barczak [SNB07]. It is an algorithm that optimises the triangle drawing order in linear time to achieve good utilisation of a FIFO vertex transform cache of a given size k, by simulating such a cache.

The general structure of the algorithm is as follows:

1. Choose any vertex as the initial so called fanning vertex.
2. Emit all triangles which use the current fanning vertex and haven't been emitted yet, by appending them to the output triangle list. All the vertices used are candidates to be the next fanning vertex and are added to a so called dead-end stack.
3. Choose the next fanning vertex among the candidates still having live triangles (triangles not yet emitted). Repeat from step 2.
4. If none of the candidates have live triangles, the algorithm has reached a dead-end. In that case, backtrack through all used vertices (by removing vertices from the dead-end stack) until one with live triangles is found, and repeat from step 2.

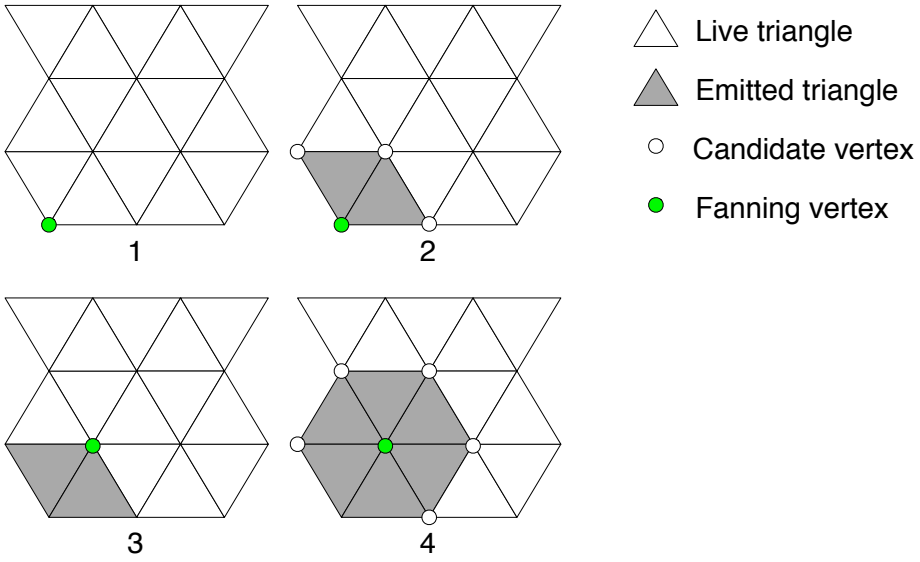

Figure 5.1: The main work flow of the tipsify algorithm. In state 1, a vertex is chosen to be fanning vertex. In state 2, all triangles using the fanning vertex are emitted, their vertices are candidates to be the next fanning vertex. In state 3, one of the candidates is chosen to be fanning vertex, and its live triangles are emitted in state 4.

5. If no new fanning vertex is found in steps 3 and 4, scan all vertices linearly until one with live triangles is found, and repeat from step 2. If none is found, terminate.

The core process in steps 1-3 above is illustrated in figure 5.1.

In step 3 above, when choosing among the candidates, the valid candidates are prioritised. The longer a vertex has been in the cache, and thus the sooner it will be thrown out from the cache, the higher priority it has. For each candidate evaluated, the number of new vertices to transform is estimated as twice the number of triangles to emit. If the cache simulation concludes that the candidate vertex would be pushed out from the cache by the new transformed vertices, the candidate is given the lowest priority.

The pseudocode for this algorithm, as presented by Sander et al., is included in listing 5.1. The algorithm takes a triangle list I and a cache size k as parameters and returns a reordered triangle list O.

Listing 5.1: Original tipsify pseudocode [SNB07]. Note that the original version had an error on line 28.

```
1  Tipsify(I, k): O
2   A = Build-Adjacency(I)          Vertex-triangle adjacency
3   L = Get-Triangle-Counts(A)      Per-vertex live triangle counts
4   C = Zero(Vertex-Count(I))       Per-vertex caching time stamps
5   D = Empty-Stack()               Dead-end vertex stack
6   E = False(Triangle-Count(I))    Per triangle emitted flag
7   O = Empty-Index-Buffer()        Empty output buffer
8   f = 0                           Arbitrary starting vertex
9   s = k+1, i = 1                  Time stamp and cursor
10  while f >= 0                    For all valid fanning vertices
11    N = Empty-Set()               1-ring of next candidates
12    foreach Triangle t in Neighbors(A, f)
13      if !Emitted(E,t)
14        for each Vertex v in t
15          Append(O,v)             Output vertex
16          Push(D,v)               Add to dead-end stack
17          Insert(N,v)             Register as candidate
18          L[v] = L[v]-1           Decrease live triangle count
19          if s-C[v] > k           If not in cache
20            C[v] = s              Set time stamp
21            s = s+1               Increment time stamp
22        E[t] = true               Flag triangle as emitted
23    f = Get-Next-Vertex(I,i,k,N,C,s,L,D)
24                                  Select next fanning vertex
25  return O
26
27 Get-Next-Vertex(I,i,k,N,C,s,L,D)
28  n = -1, m = -1                  Best candidate and priority
29  foreach Vertex V in N
30    if L[v] > 0                   Must have live triangles
31      p = 0                       Initial priority
32      if s-C[v]+2*L[v] <= k       In cache even after fanning?
33        p = s-C[v]                Priority is position in cache
34      if p > m                    Keep best candidate
35        m = p
36        n = v
37  if n == -1                      Reached a dead-end?
38    n = Skip-Dead-End(L,D,I,i)    Get non-local vertex
39  return n
40
41 Skip-Dead-End(L,D,I,i)
42  while !Empty(D)                 Next in dead-end stack
```

```
43      d = Pop(D)
44      if L[d] > 0                    Check for live triangles
45        return d
46    while i < Vertex-Count(I)        Next in input order
47      i = i + 1                      Cursor sweeps list only once
48      if L[i] > 0                    Check for live triangles
49        return i
50    return -1                        We are done!
```

This design of the algorithm makes it very fast, by emitting many triangles (all live triangles for the current vertex) between making choices, and by roughly estimating how good a choice each candidate vertex is instead of trying to choose among the candidates by investigating each choice more in detail. The fact that the algorithm is fast and runs in linear time, combined with the ability to optimise for a given cache size, makes this algorithm ideal for implementation at load time. In this way, the models could be optimised specifically for each usage context, without knowledge on these contexts beforehand.

5.2 Forsyth's reordering algorithm

In September 2006, Forsyth presented a fast, linear-speed algorithm for reordering triangles [For06]. This algorithm does not optimise for any specific cache size, but tries to generate an ordering which works well with most cache sizes.

Forsyth's algorithm simulates a cache with a least recently used (LRU) replacement policy, even though the real target hardware might use a FIFO cache. The simulated LRU cache is used to keep track of how recently used the vertices are. The size of this simulated cache need not match the size of the target hardware cache, since it is only used for estimating how good a choice different vertices are. The larger the simulated cache, the more vertices are kept track of, potentially leading to better decisions.

The algorithm gives scores for vertices as a function of how recently the vertex was used and how many triangles yet are to use this vertex. The score for triangles is calculated as the sum of the score of its vertices. The triangle with the highest score of the triangles which still have not been added is added to the output list, its vertices are moved to the front of the simulated LRU cache, the scores are recalculated and the next triangle is chosen.

This algorithm is described in pseudocode in listing 5.2, where the algorithm takes a triangle list I as a parameter and returns a reordered triangle list O.

Listing 5.2: Pseudocode for Forsyth's algorithm

```
1  ReorderForsyth(I): O
2   A = Build-Adjacency(I)          Vertex-triangle adjacency
3   L = Get-Triangle-Counts(A)      Per-vertex active triangle count
4   E = False(Triangle-Count(I))    Per-triangle added flag
5   P = MinusOne(Vertex-Count(I))   Per-vertex cache position
6   O = Empty-Index-Buffer()        Empty output buffer
7   S = Zero(Vertex-Count(I))       Per-vertex score
8   T = Zero(Triangle-Count(I))     Per-triangle score
9   C = MinusOne(CACHE_SIZE + 3)    Simulated cache
10  i = 0                           Linear scanning cursor
11
12  for each Vertex v in I
13    S[v]= FindVertexScore(v,L,P)  Initialise the vertex score
14    for each Triangle t in Neighbours(A,v)
15      T[t] = T[t] + S[v]          Add the score to the triangles
16
17  m = -1                          Best score
18  n = -1                          Best triangle
19  for each Triangle t in I        Find the best triangle
20    if T[t] > m
21      m = T[t]
22      n = t
23
24  while n >= 0                    As long as we have a triangle
25    E[n] = true                   Flag triangle as added
26    for each Vertex v in n
27      Append(O,v)                 Output vertex
28      MoveToFront(C,P,v)          Move to the front of the cache
29      RemoveTriangle(A,v,n)       Remove this triangle from v
30      L[v] = L[v] - 1             Decrease the active triangle count
31
32    for each index j in C         Update the scores in the cache
33      v = C[j]                    The vertex in this cache slot
34      if j >= CACHE_SIZE          The vertex has been pushed out
35        P[v] = -1                 Mark as not cached
36        C[j] = -1                 Remove from the cache
37      s = FindVertexScore(v,L,P)  New score
38      for each Triangle t in Neighbours(A,v)
39        T[t] = T[t] + s - S[v]    Update the score of the triangles
40      S[v] = s                    Update the vertex score
41
42    m = -1                        Best score
43    n = -1                        Best triangle
44    for each Vertex v in C        Triangles referenced by the cache
```

```
45        for each Triangle t in Neighbours(A,v)
46          if T[t] > m                 Find the best triangle
47            m = T[t]
48            n = t
49
50      if n < 0                        No active triangle in the cache
51        while i < Triangle-Count(i)
52          if E[i]                     This triangle is added
53            i = i + 1                 Skip to the next one
54          else
55            n = i                     Continue from this triangle
56            break
57
58    return O
59
60  MoveToFront(C,P,v)
61    e = P[v]                          Previous cache position
62    if e < 0                          If not in cache earlier, move
63      e = CACHE_SIZE + 2              all vertices in the cache
64    for i = e to 1
65      C[i] = C[i-1]                   Move this entry one step back
66      if C[i] >= 0
67        P[C[i]] = P[C[i]] + 1         Update the cache position info
68    C[0] = v                          Put the vertex at the front
69    P[v] = 0                          Update the cache position info
```

5.2.1 Score function

The core of Forsyth's algorithm lies in the design of the score function. The more recently used a vertex is, the higher score it should be given. The three most recently used vertices, used by the latest triangle, should be given the same score, since their internal order within the triangle can be more or less arbitrary. To avoid leaving individual triangles behind, which will be costly to add later, the scoring must encourage using vertices which will be needed by only a few more triangles. Therefore, a low number of active triangles (triangles not yet added) for a vertex should also increase the score.

The score function proposed by Forsyth therefore is of the following form:

$$f(p,a) = \begin{cases} -1 & \text{if } a = 0 \quad \text{(no triangles need this vertex)} \\ f_p(p) + f_a(a) & a > 0 \quad \text{(otherwise)} \end{cases}$$

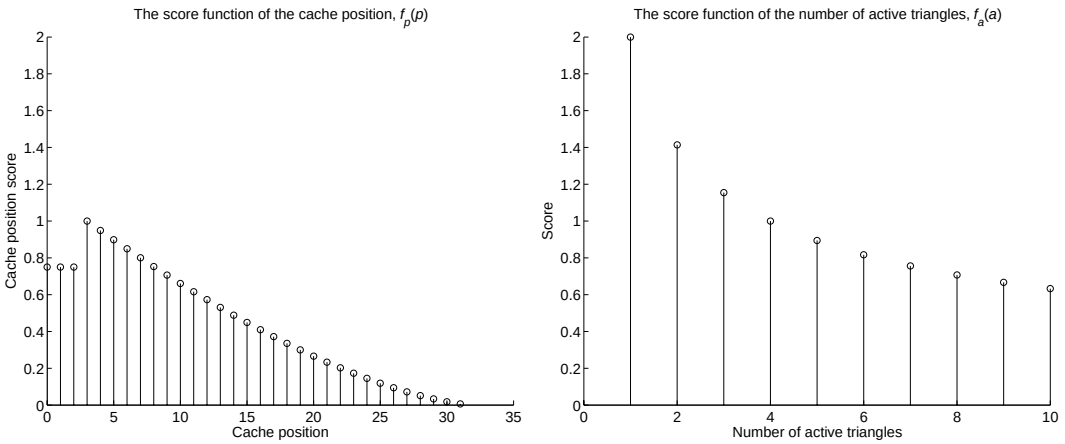

Figure 5.2: The score functions in Forsyth's algorithm.

$$f_p(p) = \begin{cases} 0 & p < 0 \quad \text{(not in the cache)} \\ A & p < 3 \quad \text{(used by the latest triangle)} \\ \left(1 - \frac{p-3}{S-3}\right)^B & p \geq 3 \quad \text{(otherwise)} \end{cases}$$

$$f_a(a) = C \cdot a^{-D}$$

The parameter p is the cache position and a is the number of active triangles.

Forsyth did not see any improvements in using a larger simulated cache than $S = 32$ and therefore chose that size. By varying the parameters A, B, C and D and testing the result with simulations, he concluded that the optimal parameters were

$$\begin{cases} A = 0.75 \\ B = 1.5 \\ C = 2.0 \\ D = 0.5. \end{cases}$$

The two components of the score function are illustrated in figure 5.2.

Even though these score functions may seem computationally heavy, they need not be calculated every time they are used. $f_p(p)$ is only used for values of p from 0 to one less than the cache table size. The parameter a in $f_a(a)$ can in principle be any positive value but is in practice usually a small value, less than 7 in most cases. By cutting off the tail of the function $f_a(a)$ for all values of a larger than an arbitrary threshold, both of these functions can be precalculated and stored in a table, avoiding heavy calculations at algorithm run time. To minimise the impact of cutting the tail of $f_a(a)$, the threshold must be higher than the values of a where the value of $f_a(a)$ actually impacts the decisions in the algorithm.

Recall that the purpose of $f_a(a)$ is to prioritise vertices with few active triangles left. In the tests, the threshold was chosen to be 32.

5.3 View independent overdraw reduction

When reducing overdraw, the completely visible triangles are drawn first and the more occluded ones later. Visibility depends on where the object is viewed from, though. Therefore, to avoid overdraw, the triangles would have to be reordered specifically for each frame rendered. However, Sander et al. presented a technique for sorting clusters of triangles to get overdraw reduction independent of viewing direction [SNB07].

In this technique, each cluster is assigned an occlusion potential, where a higher potential means a higher probability of occluding other triangles, independent of views. The occlusion potential of a cluster P in the mesh M is approximated by

$$\mathscr{O}'(P, M) = (C(P) - C(M)) \cdot N(P),$$

where C is the centroid function and N is the average normal of the cluster P. This approximation simply says that clusters far from the mesh centre, facing away from the mesh centre are the ones most probably occluding other parts of the mesh, regardless of from which direction it is viewed.

This is a reasonable approximation, assuming that the mesh is to be viewed from the outside and not from the inside, and assuming that the mesh consists of one single part where the centre is inside of the mesh.

For the average normal of the clusters to be as useful as possible, the average normal should be representative for the whole cluster, that is, the normals within the cluster should not vary much.

In order not to compromise the vertex cache efficiency, the mesh can be divided into clusters during vertex cache optimisation. During the triangle reordering, an estimate of the current ACMR is calculated. When the estimate drops below a given threshold value, named λ in Sander's original description, the triangles emitted since the last cluster are considered to be a new cluster. Then the ACMR estimation is reset and the algorithm continues emitting triangles. In this way, every cluster except the last one has an average estimated ACMR equal to λ, regardless of the order the clusters are drawn. Therefore, the clusters can be reordered freely without affecting the total ACMR.

The same method can equally well be integrated into Forsyth's reordering algorithm, by using another estimate of the ACMR.

Chapter 6

Memory usage

The memory usage for the algorithms optimising the vertex cache usage is analysed below, since memory is usually a very limited resource in embedded systems. Conversely, reducing the memory usage makes the algorithms usable on larger models, both on embedded systems and on desktop computers.

The memory usage of the view independent overdraw reduction algorithm is not analysed in detail, since it is trivial. It uses a certain amount of memory for each cluster, but the memory usage can easily be limited by limiting the number of clusters.

The analysed algorithms require memory for the temporary buffers used while reordering triangles. Below, t is the number of triangles and v is the number of vertices. In the description of tipsify, k is the tipsify cache size parameter.

The memory usage of the initial mesh can be used to estimate proportions to the sizes discussed later. A mesh consisting of vertex positions and texture coordinates requires 5 coordinates per vertex, assuming that 3 coordinates are used for the positions. If each coordinate is given with 16 bits precision, the vertex data is is $10v$ bytes. If the triangles are defined as triangle strips with an IPT ratio of 1.6, the triangle data occupies $3.2t$ bytes if 16 bit indices are used. Applying the simplification $t \approx 2v$, the vertex data becomes $5t$ bytes, making the total of the mesh $8.2t$ bytes. As an example of a mesh requiring more memory, add normal vectors to the vertex data and use triangle lists instead of strips. In this case, each vertex needs 8 coordinates, making the vertex data part $8t$ bytes. The triangle lists need $6t$ bytes, giving a total of $14t$ bytes.

6.1 The tipsify algorithm

The memory usage of the tipsify algorithm, as summarised in table 6.1, can be adjusted by using data types of different sizes. For analysing the size of the data structures, four primitive data types need to be defined. AdjacencyType is a nonnegative integer capable of storing the number of triangles one vertex belongs to. VertexIndexType is a nonnegative integer, an index to a vertex. This is the data type used for the index buffers given to and returned from the algorithm. TriangleIndexType is a nonnegative integer for storing triangle numbers. ArrayIndexType is a nonnegative integer for storing indices into the index buffers, that is, it must be able to store values up to $3t$. In the usage described below, ArrayIndexType must actually support values up to $3t + k + 1$.

The adjacency data structure named A in the pseudocode (listing 5.1) is implemented as an array of triangle indices which maps vertices to the triangles using them, and another array with offsets into the first one. In the reference C++ implementation in appendix A, these are named adjacency and offsets, respectively. Every vertex can be used by a variable number of triangles, therefore the number of triangle indices for each vertex is not fixed. But since each triangle uses three vertices, the total amount of triangle references in the array will be $3t$. The offsets array contains offsets into the adjacency array, pointing to the first triangle index for each vertex. In order to construct these arrays, another array (numOccurrances) is used, containing the number of triangles for each vertex. After building the adjacency data structures, this array is reused as liveTriangles (L in the pseudocode).

The array of time stamps (C, cacheTime) contains one time stamp for each vertex. The array is initialised to zero, and the time starts at $k + 1$ and is increased by one unit for each cache miss. In the worst case, every vertex access is a cache miss, which gives the maximum time stamps value $k + 1 + 3t$. ArrayIndexType is the type used for this array.

The dead-end stack (D, deadEndStack) contains vertex indices. In the worst case, the algorithm never runs into any dead-ends, and no indices are popped off the stack. In that case, the dead-end stack grows to a size of $3t$.

The array of flags keeping track of which triangles are emitted (E, emitted) can be efficiently stored as an array of $\lceil t/8 \rceil$ bytes, with one bit for each triangle.

The output index buffer (O, outputIndices) will contain $3t$ vertex indices.

Additionally, the set of candidates to the next vertex choice (N, nextCandidates)

Name	Pseudocode name	Type	Length
adjacency	A	TriangleIndexType	$3t$
offsets	A	ArrayIndexType	v
numOccurrances	A	AdjacencyType	v
liveTriangles	L		
cacheTime	C	ArrayIndexType	v
deadEndStack	D	VertexIndexType	$3t$
emitted	E	byte	$\lceil t/8 \rceil$
outputIndices	O	VertexIndexType	$3t$
nextCandidates	N	VertexIndexType	3·maxAdjacency

Table 6.1: Data structures, their type and memory usage in the tipsify algorithm

will at most contain three times the number of triangles using the current vertex. The highest number of triangles referring one single vertex (maxAdjacency) can easily be obtained while building the adjacency data structures. There is also an upper bound for this, the maximum value of AdjacencyType.

In order to estimate the total amount of memory needed, the sizes of the data types must be defined. As an example, consider that vertex indices (VertexIndexType) are constrained to 16 bit by the environment, as in OpenGL ES 1.x. The number of triangles is not externally constrained, and can exceed 16 bits, therefore 32 bits are needed for this.

In most normal meshes one vertex is only used by a handful of triangles, and very seldom by more than 255 triangles, and thus AdjacencyType can quite safely be limited to 8 bits. If a mesh contains a vertex used by more than 255 triangles, that mesh can be reordered by a less optimised version of the algorithm.

The memory used by nextCandidates is almost constant and can be disregarded, since maxAdjacency does not grow significantly with the mesh size, and it additionally is upper bound by the maximum value of AdjacencyType. With these definitions and simplifications, the memory needed is

$$4 \cdot 3 \cdot t + 4 \cdot v + 1 \cdot v + 4 \cdot v + 2 \cdot 3 \cdot t + 1 \cdot t/8 + 2 \cdot 3 \cdot t =$$

$$24.125t + 9v$$

bytes, which can be further simplified by introducing the approximation $t \approx 2v$, yielding

$$24.125t + 4.5t = 28.625t$$

bytes.

To lower the peak of memory usage, one simple modification can be introduced. Instead of directly outputting each vertex into the output buffer, the output order can initially be stored only as triangle indices. When the output order is complete, the rest of the temporary buffers can be freed and the real output buffer can be allocated. The original outputIndices array is of type VertexIndexType and $3t$ elements long, requiring $6t$ bytes in the example case, but can in this case be replaced with an array of type TriangleIndexType which is only t elements long, requiring $4t$ bytes. Thus, the amount of memory saved is $2t$ bytes.

To further lower the memory usage, the number of triangles could be limited to a 16 bit number, making TriangleIndexType a 16 bit type. That would save $6t$ bytes and another $2t$ bytes if combined with the optimisation above. If these optimisations are combined, the total memory usage is $18.625t$ bytes.

6.2 Forsyth's reordering algorithm

The memory usage of Forsyth's algorithm is shown in table 6.2. As in the analysis above, some primitive data types need to be defined. AdjacencyType, VertexIndexType and TriangleIndexType are defined in the same way as for tipsify. In addition to these, a signed integer type CachePosType for storing cache positions is needed. It must be capable of storing values up to the cache size S plus two (since the cache array contains three extra slots) and the value -1 representing a vertex not in the cache. ScoreType is a data type for the vertex and triangle scores. This type can be defined freely as either integer or floating point, since the score values can be scaled to use the whole data type range if it is an integer type.

The adjacency data structure, active triangle count, triangle added flags and output index buffer are implemented in the same way as in tipsify, but they are named triangleIndices, offsets, numActiveTris, triangleAdded and outIndices. Additionally, the current score (lastScore, S) and cache position (cacheTag, P) is stored for each vertex, and the current score (triangleScore, T) for each triangle. The vertex index (cache, C) is stored for each simulated cache slot (and for three additional slots, needed for keeping track of old vertices pushed out of the cache) but since the simulated cache is fixed and does not grow with the mesh, this is omitted from the following calculations.

Name	Pseudocode name	Type	Length
triangleIndices	A	TriangleIndexType	$3t$
offsets	A	ArrayIndexType	v
numActiveTris	A/L	AdjacencyType	v
triangleAdded	E	byte	$\lceil t/8 \rceil$
outIndices	O	VertexIndexType	$3t$
lastScore	S	ScoreType	v
cacheTag	P	CachePosType	v
triangleScore	T	ScoreType	t

Table 6.2: Data structures, their type and memory usage in Forsyth's algorithm

To get comparable memory usage values, assume the same limitations and definitions as for tipsify, that is, VertexIndexType is 16 bits, TriangleIndexType and ArrayIndexType are 32 bits and AdjacencyType 8 bits. As long as the cache is less than or equal to 125 elements large, CachePosType can be a signed 8 bit type.

If ScoreType is chosen to be an integer, a suitable scaling must be selected, mapping the whole score function value range into the range of the chosen integer type. The score function returns values in the range [-1, 3], but returns negative values only if the vertex is not needed by any active triangles. The actual score value for that case does not affect the reordering result, since the score will be added only to triangles which are not active and the scores of those triangles are not used anywhere. Therefore, the score function can be modified to return 0 in that case, making the score function nonnegative for all input values and shrinking the value range to [0, 3]. By making the value range smaller, an integer representation of the score function can use larger scaling factors, giving better precision.

The modified score function gives values in the range [0, 3], therefore the triangle score lies in the range [0, 9]. The maximum triangle score actually is slightly lower, since a triangle with three unique vertices cannot get the maximum vertex cache score for all three vertices. If an unsigned 16 bit integer type is chosen, the score values can be scaled by a factor up to $\lfloor 65535/9 \rfloor = 7281$ before conversion to integers. For 8 bit integers, the maximum scaling factor would be $\lfloor 255/9 \rfloor = 28$. The following calculations assume that ScoreType is a 16 bit integer.

With these definitions, the memory needed for Forsyth's algorithm is

$$4 \cdot 3 \cdot t + 4 \cdot v + 1 \cdot v + 1 \cdot t/8 + 2 \cdot 3 \cdot t + 2 \cdot v + 1 \cdot v + 2 \cdot t =$$

$$20.125t + 8v$$

bytes, which by the approximation $t \approx 2v$ is simplified to $24.125t$ bytes.

By changing ScoreType to a 8 bit integer, the memory usage would be reduced by $1.5t$ bytes. The two other general optimisations proposed for tipsify, changing TriangleIndexType to a 16 bit type and only storing triangle indices (instead of the final vertex indices) during reordering can be applied here, too.

Changing TriangleIndexType to a 16 bit type saves $6t$ bytes, storing triangle indices instead of full triangles saves $2t$ bytes and yet another $2t$ bytes combined with a 16 bit TriangleIndexType. If these optimisations are combined, the total memory usage is $12.625t$ bytes.

Chapter 7

Test setup

7.1 Test environments

For testing, verification and evaluation, the algorithms were implemented both on desktop/laptop platforms using OpenGL and on mobile platforms using OpenGL ES. In order to verify the feasibility and to evaluate the benefit of integration into existing APIs, the algorithms were integrated into Nokia's M3G library.

Two different laptops were used as OpenGL test environments. The first one was an Apple Powerbook with a NVIDIA GeForce FX Go5200 64 MB chipset, running Mac OS X 10.5.1. The second one was a Lenovo Thinkpad T60p with an ATI Mobility FireGL V5200 512 MB chipset, running Windows XP SP2 and the OpenGL driver version 6.14.10.5527. For testing in mobile environments, three smartphones were used. Nokia E65 and E51 phones, having Nokia OpenGL ES 1.0 and 1.1 respectively, were used for testing software renderers, and a Nokia E90 with a PowerVR MBX chip for testing hardware renderers. Even though there is only a small difference in version number between Nokia OpenGL ES 1.0 and 1.1, they differ distinctly in behaviour in some tests.

Note that the laptop chipsets were not chosen to be compared against each other. The NVIDIA chip is much older and not aimed at the same performance range as the ATI chip. They are only used for analysing the behaviour of and relative improvement offered by different algorithms.

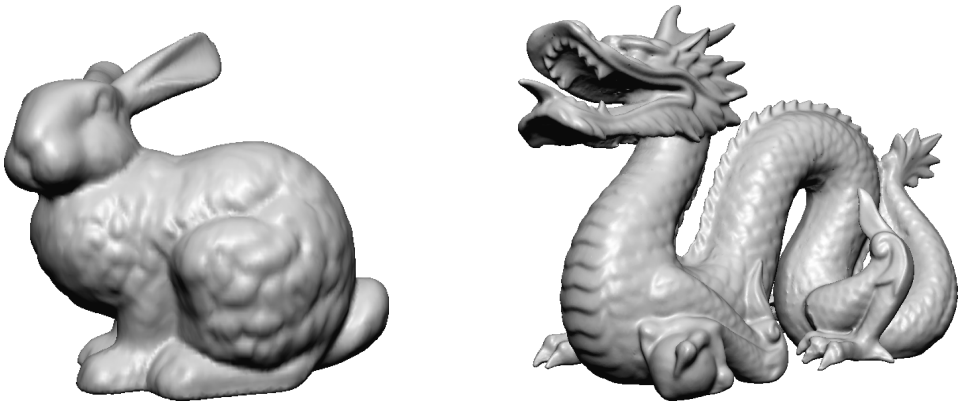

Figure 7.1: The bunny and dragon models from Stanford.

Figure 7.2: M3G test models. From the left, hi_dog, grid, mapanimlow and texturedcar.

7.2 Test models

For measuring ACMR and testing the maximum possible gain from reordered triangles, large meshes from the Stanford 3D Scanning Repository were used. The bunny model (with over 69 000 triangles) was used in both environments, and the dragon model (with over 1 000 000 triangles) was used in the laptop environments. These models are shown in figure 7.1. Note that these models are automatically constructed from scanning data, using two different algorithms. Since they have been constructed automatically, their topology is probably slightly more regular than hand-modelled meshes.

The M3G integration was tested with a few different models, shown in figure 7.2. The first M3G model, hi_dog, is a complex animation with dynamic lighting, including a large skinned mesh, created by HI Corporation. The grid model is a collection of 12 application icons from the Nokia S60 application menu, with simple animation by rotating whole meshes. The mapanimlow mesh is a car race animation with low resolution textures and without lighting, from the JBench-

	IPT	Submeshes	Vertices	Triangles
bunny1-opt	1.85	1	35947	69451
bunny1	6.00	1	35947	69451
bunny2-opt	2.01	1	8171	16301
bunny2	6.00	1	8171	16301
bunny3-opt	2.11	1	1889	3851
bunny3	6.00	1	1889	3851
bunny4-opt	2.17	1	453	948
bunny4	6.00	1	453	948
grid	2.13	101	4039	3419
hi_dog	2.78	30	14306	20240
mapanimlow	2.13	132	3506	4322
texturedcar	2.02	2	6692	10216

Table 7.1: Test model information

mark Pro application, made by Kishonti Informatics LP. The texturedcar model is a static, non-animated version of the race car in higher resolution.

In addition to these models with original M3G content, the Stanford bunny was converted into M3G format. The bunny model is available in four resolutions, the original one (bunny1) and three decimated versions with less vertices and triangles (bunny2, bunny3 and bunny4). The M3G file format only supports triangles in strip format. Each version of the mesh was exported to M3G both as a mesh with the triangles in exactly the same order as in the original version, with separate strips for every individual triangle, and as a mesh with the triangles reordered into strips using a trivial algorithm (file names with an -opt suffix).

Statistics on the M3G models are shown in table 7.1.

Chapter 8

The behaviour of the tipsify algorithm

8.1 Simulation results

To study the behaviour and properties of the tipsify algorithm, the output from the algorithm was simulated with a theoretical pure FIFO cache, to calculate the ACMR for the orderings when using such a cache. The ACMR as a function of the cache size given as parameter to tipsify and the simulated cache size is illustrated in figure 8.1.

The surface plot shows exactly what can be expected. The diagonal, where tipsify was given the same size as used for the ACMR simulation, divides the graph into two distinct areas. The lower left area is the area where tipsify was given an underestimate of the target cache, the upper right area is where tipsify was given an overestimate. The behaviour can be inspected more in detail in the other two graphs, showing cross-cuts of the surface from different points.

The first of the two cross-cut graphs shows how a specific tipsify output works on different caches. When the real cache is smaller than the tipsify target size, the ACMR is very high, but falls steeply when the cache size approaches the intended size. With a cache larger than the tipsify target size, the ACMR hardly improves any further at all.

The second cross-cut graph shows the case when the simulated cache is of a fixed size and tipsify is given varying sizes as a parameter. When tipsify is given a target smaller than the real size, the results are slightly suboptimal, but mostly acceptable. However, as soon as tipsify is given a larger target than the real cache

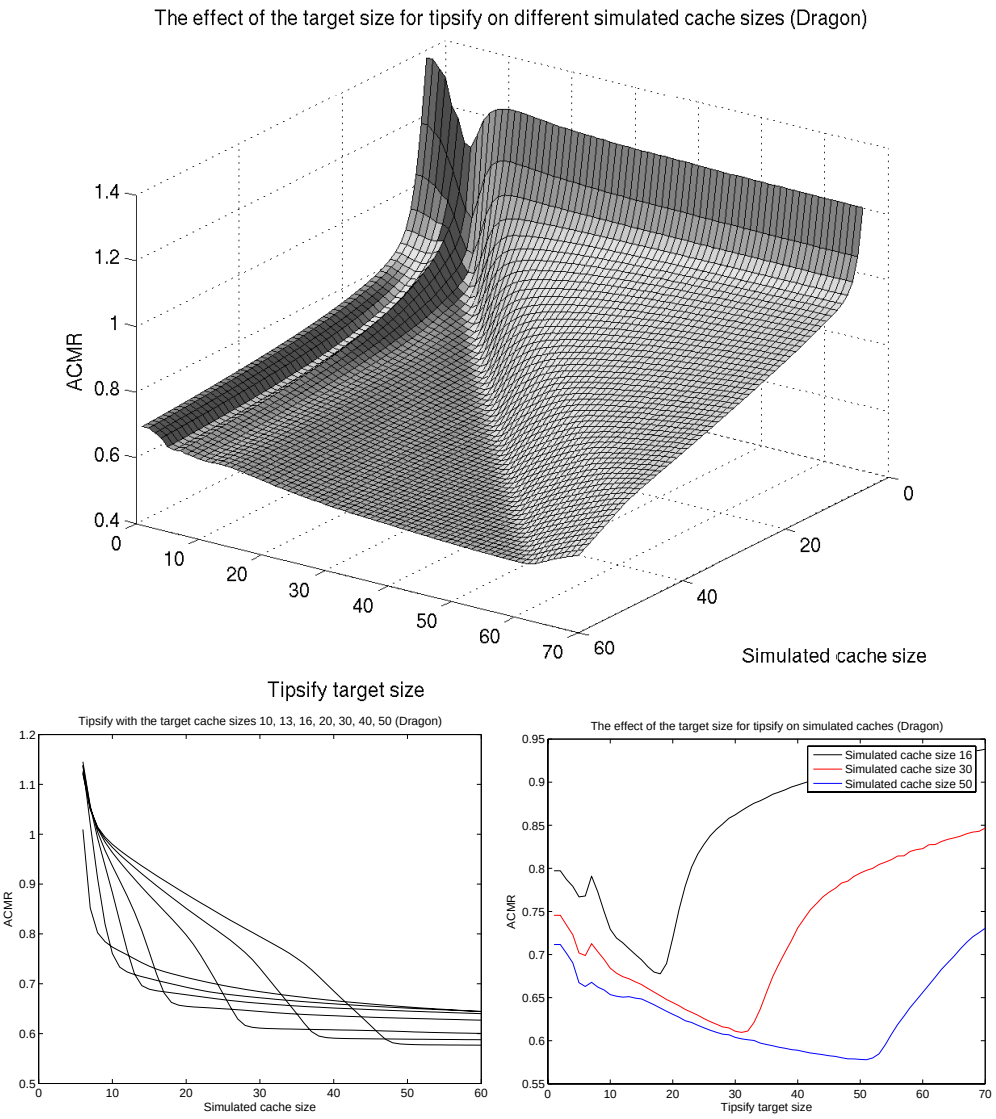

Figure 8.1: General properties of tipsify. The surface plot shows how the ACMR varies with the size given to tipsify and the actual cache size. The other two plots show how a given tipsified ordering performs on different cache sizes, and how different tipsify target sizes perform on fixed cache sizes. The simulations were made on the dragon model, the main features are similar for other models.

size the ACMR becomes much worse. Therefore, an overestimation of the cache size should be avoided.

8.2 Choosing the optimal tipsify parameter

One of the main features of tipsify, the ability to optimise for a specific cache size, is at the same time its main drawback. In practice, the cache size of the target device is not necessarily known. (Since the cache is completely transparent, current graphic APIs such as OpenGL and OpenGL ES do not even have any notion of a vertex cache, and thus querying for its size is impossible.) Additionally, knowledge of the actual cache size may not be enough, since the actual cache is not necessarily a pure FIFO cache. If this is the case, the optimal tipsify parameter is not likely the real cache size, since tipsify assumes that the cache is a pure FIFO cache and simulates a such.

Therefore, either more specific knowledge about the underlying layer (hardware chip or software API implementation) or a run time test to determine the optimal parameter is needed.

This can be tested simply by benchmarking the order output by tipsify for a range of cache sizes, and finding the shortest rendering time. This was tested on a few phones with different renderers (two software renderers and one hardware accelerated renderer) and on two laptops. The rendering times from these tests are illustrated in figure 8.2.

If the actual cache is a pure FIFO cache, the rendering time curve should have the same shape as the simulated ACMR curves, assuming that the rendering time consists of one part of a fixed length and another part which varies linearly with the ACMR.

The rendering time curves for the two software renderers (on E51 and E65) have the same general features as the simulated curves, but on E51 there is no clear optimum with a steep rise in ACMR afterwards, as in the simulations. On E65 the rise in ACMR is not as clear and steep as in the simulations, but still more clear than on E51. This indicates that the cache is not a pure FIFO in either of the cases.

The rendering times on E90 show an optimum around the parameter value 6. With higher parameter values the rendering times are slightly longer and most of the differences in that area are measurement noise. Note that the relative sizes

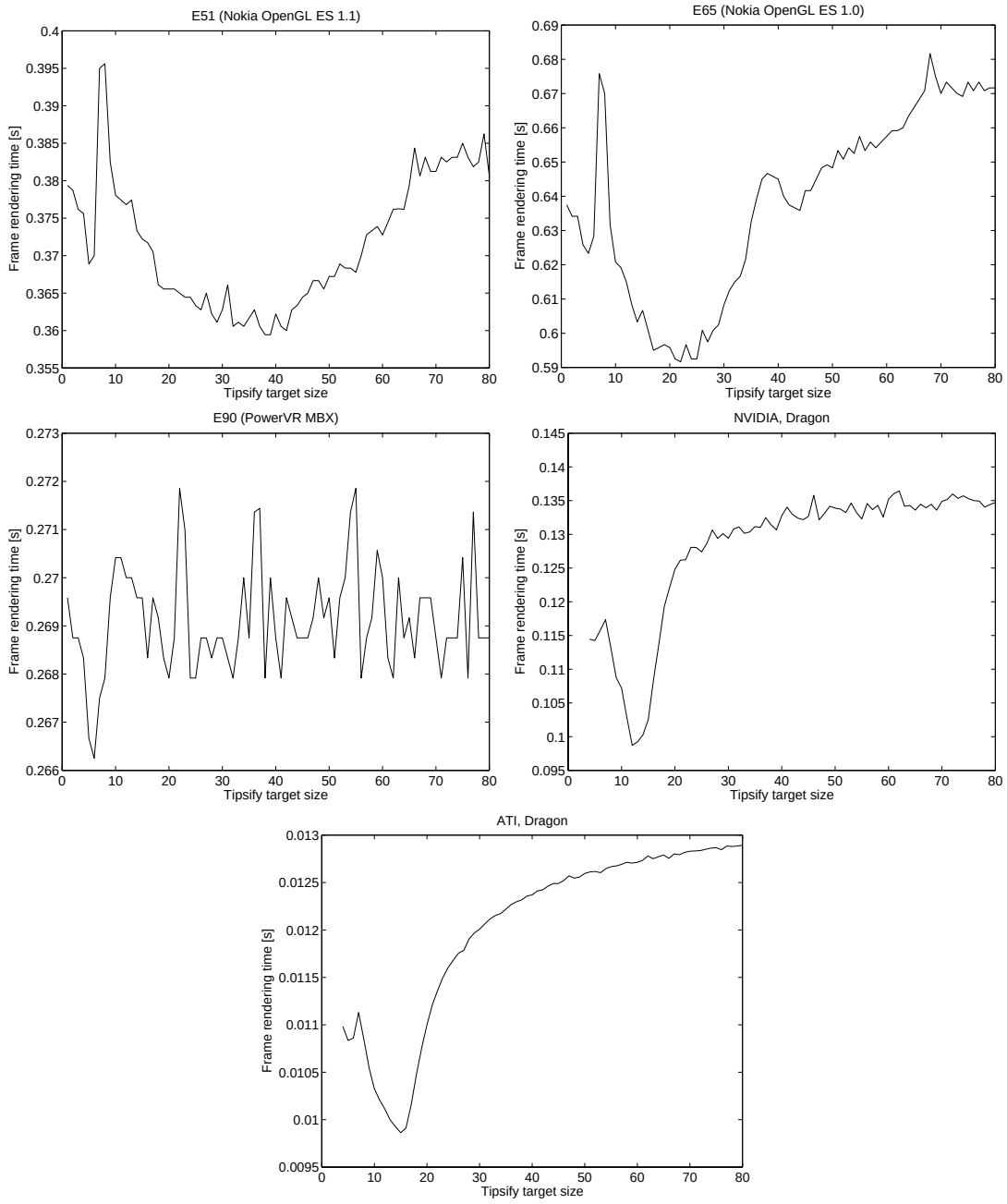

Figure 8.2: Rendering times of a model reordered with different tipsify target sizes, on different OpenGL and OpenGL ES implementations. The mobile platforms were tested with the bunny model, the laptop chipsets with the dragon model. The rendering times are measured in seconds.

of the variations are small and not significant in practice. The conclusion is that the actual cache is probably very small.

The shape of the graphs for both laptop chipsets (NVIDIA and ATI) is almost identical to the simulated curves, meaning that they probably have more or less ideal FIFO caches of about 16 elements. Choosing the optimal parameter is very easy in these cases.

Chapter 9

Algorithm improvements and compromises

The vertex cache optimisation algorithms, tipsify and Forsyth's algorithm, were analysed for potential improvements in memory usage, run time performance and ACMR results. The improvements made are explained below.

9.1 The tipsify algorithm

9.1.1 Dead-end stack

All of the data structures listed for tipsify in section 6.1, except the dead-end stack, are fully used. For the dead-end stack, only a worst case estimate is available, which is used as its size. In practice, only about half of it is used.

To avoid this waste of memory, a small buffer, which grows when needed, could be allocated. However, reallocating memory buffers when the algorithm has started may be troublesome or impossible in some cases or environments.

This raises the question of the necessity of the dead-end stack at all. The stack provides a path for backtracking to earlier vertices when the algorithm has run into a dead-end, allowing the algorithm to continue from a position as near the latest vertices as possible. If a vertex with live triangles is found close to the dead-end, that vertex (and possibly some of its neighbours) may still be in the cache. But if the algorithm is forced to backtrack further, it does not matter at all where the algorithm continues, because the vertices it needs are not cached anymore. If no vertex with live triangles is found in the stack, the algorithm continues scanning linearly through all vertices.

The algorithm may work almost as well without any dead-end stack. Since it does not backtrack using the stack at all in that case, the restart positions will proceed linearly through the vertices. If the vertices are ordered roughly according to locality (that is, vertices with a small difference in indices are located near each other) and the algorithm reaches dead-ends often, the restart positions are located near regions which have been processed recently, and thus the result may not be much worse than the original result using a dead-end stack. On the contrary, if the vertices are ordered completely randomly, the restart positions will also be completely random. Thus, removing the dead-end stack makes the result more dependent on the original input order.

As a compromise, the dead-end stack can be of a fixed, small size N. The stack would then work roughly as a ring buffer, storing only the N latest vertices. Adding vertices to it when it is full replaces the oldest vertex in the buffer with the new one. In this way, short steps backwards in the stack are still possible, while longer traversals backwards in the stack are avoided, reverting to the linear scanning instead. Short steps backwards are the ones which in practice find vertices which still are in the cache, while the longer traversals backwards do not give any better restart positions than the linear scanning.

Even if the dead-end stack is used, the algorithm does not terminate until the whole mesh has been scanned linearly, otherwise it could miss isolated parts of the mesh. Removing or shortening the dead-end stack may therefore slightly improve the run time performance of the algorithm in addition to decreasing the memory usage.

Simulations of these variations are illustrated in figure 9.1. The simulations confirm the properties that were deduced above. Removing the dead-end stack completely affects the end result negatively, the ACMR increases by almost 0.03. Randomising the input order affects the result more when the dead-end stack is removed than in the original setup, increasing the ACMR by another 0.02.

The compromise with a fixed-size dead-end stack of 128 elements turns out to work very well, though. The result is only slightly worse than the original, the ACMR increases about 0.01. The impact of randomised input order is much smaller than when using no dead-end stack at all, increasing the ACMR by 0.006. In the original setup, with an unlimited dead-end stack, a randomised input order increases the ACMR by only 0.001.

Removing the full sized dead-end stack from the earlier memory usage calculations saves $6t$ bytes. A fixed-size dead-end stack only adds a small constant amount to

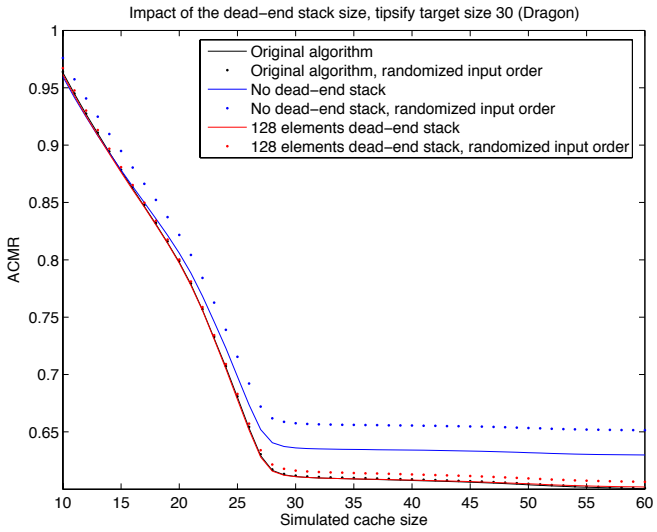

Figure 9.1: Simulation of the impact on the ACMR of removing or shrinking the dead-end stack, when using a tipsify target size of 30.

the usage sum. If this is applied to the original tipsify memory usage in section 6.1, the memory usage is lowered from $28.625t$ bytes to $22.625t$ bytes, a 20% decrease. If the other optimisations mentioned are used, it lowers the memory usage from $18.625t$ bytes to $12.625t$ bytes, which is an over 30% decrease.

Therefore, a fixed-size dead-end stack is a general memory usage improvement which does not affect the end result noticeably in practice, and can be recommended for use in all cases where the memory usage is restricted.

9.1.2 Estimating the number of uncached vertices

As seen in figure 8.1, the algorithm gives unexpected results for parameters in the region 5-10. This kind of unexpected results also appears for the bunny model, where parameter values 7 and 8 give drastically worse results than neighbouring parameter values, as illustrated in figure 9.2. The differences between the models are probably due to them being generated by two different algorithms, giving them a slightly different topology.

An explanation can be found by analysing the algorithm design. The inherent design in the tipsify algorithm assumes that the cache is not very small. The only place where the cache size given as a parameter to the algorithm actually affects the algorithm is in line 32 in the pseudocode in listing 5.1. The condition in that line, which checks whether the evaluated vertex is in the cache after adding all its live triangles, hardly ever is true for very small caches. If this condition never

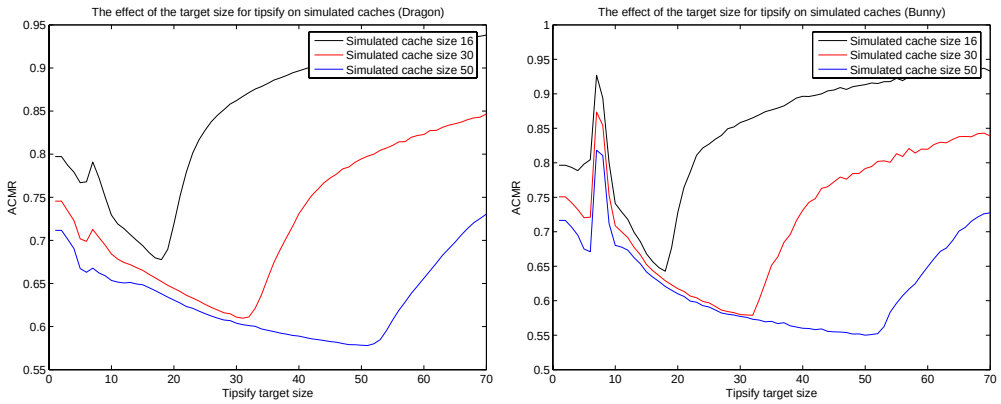

Figure 9.2: Comparison of the behaviour of tipsify between the dragon and the bunny models.

is true, the next fanning vertex is always the first valid one, whichever it happens to be.

This condition requires a heuristic estimate of the number of uncached vertices which need to be transformed when all live triangle are emitted. The original estimate used is the worst case, that every emitted triangle needs two new uncached vertices. The difference between this estimate and the real number has a minor impact on the end result as long as the cache size is large, but has a much larger impact when the cache size is small.

The estimate can of course be improved by exploring the data structures further, trying to determine the actual number of uncached vertices. That would, however, increase the algorithm run time. Instead, a few other estimates can be considered.

Figure 9.3 shows the original worst case estimate, and three other estimates. The original estimate, estimate 1, is that each new triangle requires two new uncached vertices, $2L_v$, where L_v is the number of live triangles for vertex v. Another plausible case is that each triangle requires two new vertices, except the first and last triangle which use one cached vertex each, giving estimate 2, $2L_v - 2$. In most cases in practice, most of the uncached vertices are located consecutively. In that case, each live triangle shares uncached vertices with the live triangles on each side, giving estimate 3, $L_v + 1$. The first and last of the live triangles might also share a cached vertex each with the already emitted triangles, yielding estimate 4, $L_v - 1$.

These four estimates were tested in the implementation of tipsify, and the output triangle orders were simulated with cache sizes 3, 4 and 30. The simulation results are compared in figure 9.4.

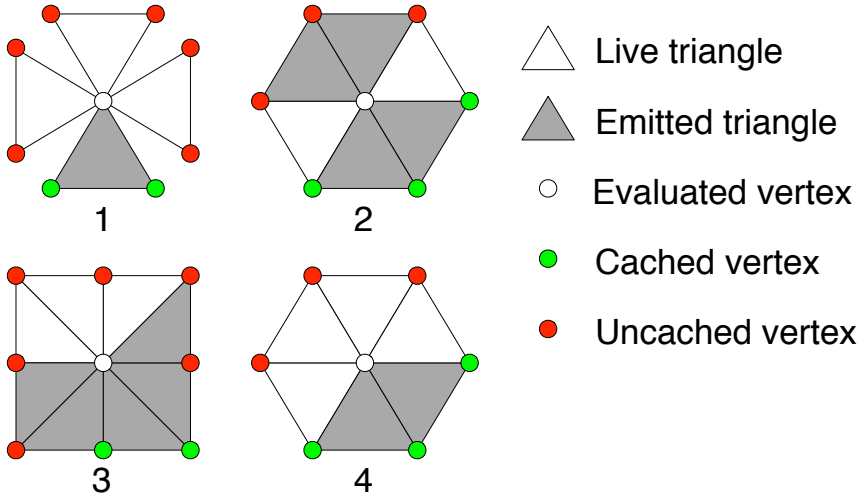

Figure 9.3: Different possible cases when estimating the number of new uncached vertices needed when evaluating a vertex. (1) $2L_v$ (2) $2L_v - 2$ (3) $L_v + 1$ (4) $L_v - 1$

For the both models, the best ACMR achieved is improved only slightly, and only for the smaller cache sizes. The point of the optimum for each cache size is moved. Notable is that estimate 4 is the only estimate yielding the best ACMR when tipsify is given the actual cache size, for the small caches. That is, the optimal tipsify parameter is 3 when simulated with a cache of size 3.

Another feature to notice is that for larger cache sizes, estimate 4 is the only one which completely avoids the suboptimal results for parameters around 5-10 on the bunny model. On the dragon model, none of the estimates give clearly suboptimal results for these parameters, but all three new estimates give better results for parameters in this range.

For these larger cache sizes, however, the optimal tipsify parameter is slightly smaller than the actual cache size (28 or 29 would be the optimal tipsify parameters for cache size 30 in the simulations), if estimate 4 is used.

The new estimates improve the behaviour when using tipsify parameters below 10, even though the tipsify algorithm is not intended for these cases. General strip building algorithms probably work better than tipsify when targeting small caches.

None of the estimates is universally the best, but estimate 4 has very small disadvantages. As long as the tipsify parameter actually is tested empirically instead of directly using exactly the actual cache size, estimate 4 can be recommended to be used instead of the original estimate, since it has a better overall behaviour.

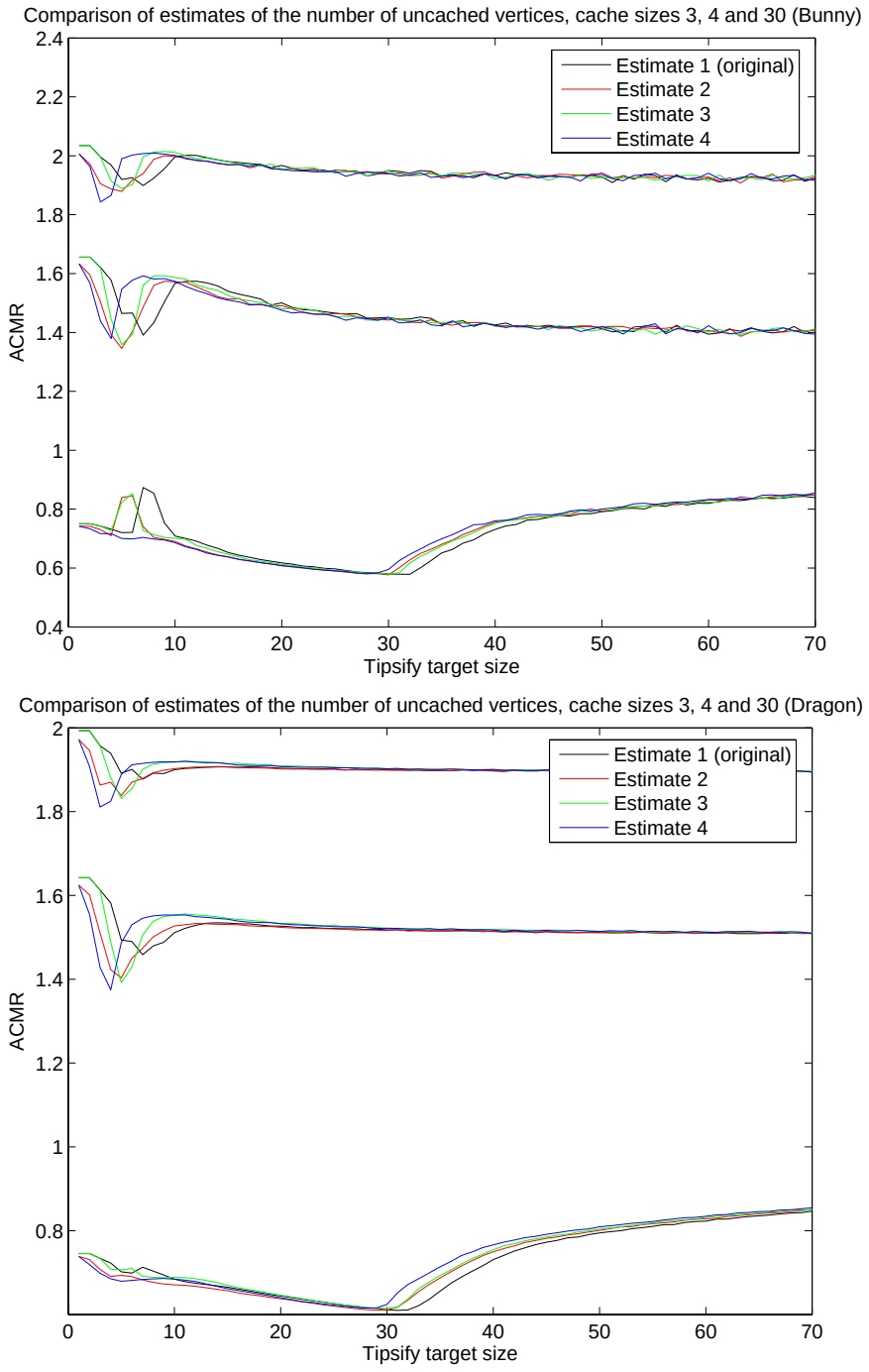

Figure 9.4: Comparison of the four estimates of the number of uncached vertices. The four algorithm variants are simulated with cache sizes 3 (the topmost curves), 4 (the curve cluster in the middle) and 30 (the bottommost curves). The top graph shows simulations on the bunny model, the bottom graph is for the dragon model.

9.2 Forsyth's reordering algorithm

9.2.1 Shrinking the cache table

The run time of large parts of Forsyth's algorithm is proportional to the size of the simulated cache. When adding vertices to the cache, the rest of the cache must be pushed back, and after adding a triangle, the scores of all vertices in the cache are updated. Therefore, the run time could be shortened by simulating a smaller cache.

The original point in simulating an LRU cache was to keep track of how recently used all vertices of interest are. By intentionally simulating a smaller cache, the algorithm deviates slightly from the original idea, but may work well enough anyway.

One potential problem lies in the cache position score function, though. The function was designed to approach zero at the end of the cache range. This is a reasonable design as long as the simulated cache is large enough and thus the benefit from reusing a vertex in the end of the cache is small. But if the simulated cache is knowingly smaller than the potential caches on the target devices, it makes little sense to give scores close to zero for the last elements in the cache, since they most probably still are cached on the real device, too, and would be a good choice for reuse. Therefore, the cache score function should not approach zero towards the end of the cache if the simulated cache is small.

When using a cache of a different size, the parameters A, B, C and D should also be reevaluated and adjusted. But simply adjusting the parameters does not change the fact that the cache score function approaches zero to the end of the cache.

By decoupling the parameter S in the score function from the actual size of the cache simulated by the algorithm, the score function values for the original cache of size 32 can be used with smaller caches, too. The cache score function in the range used by a cache of size 8, for $S = 32$ as originally and a for $S = 8$, is illustrated in figure 9.5.

Another way of presenting the argument is that a vertex in LRU cache slot n should always be given the same cache score, based on the fact that it was used n steps ago, regardless of at what point a score function converging towards zero is cut off.

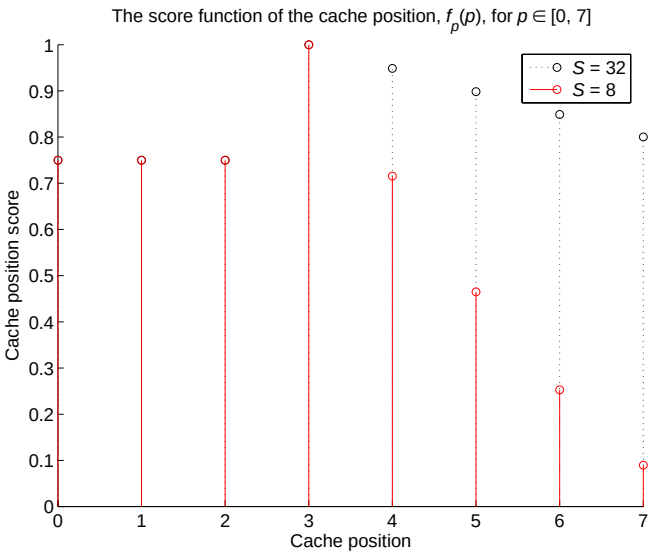

Figure 9.5: The cache score function for $S = 32$ and $S = 8$, in the range used when simulating a cache with 8 slots.

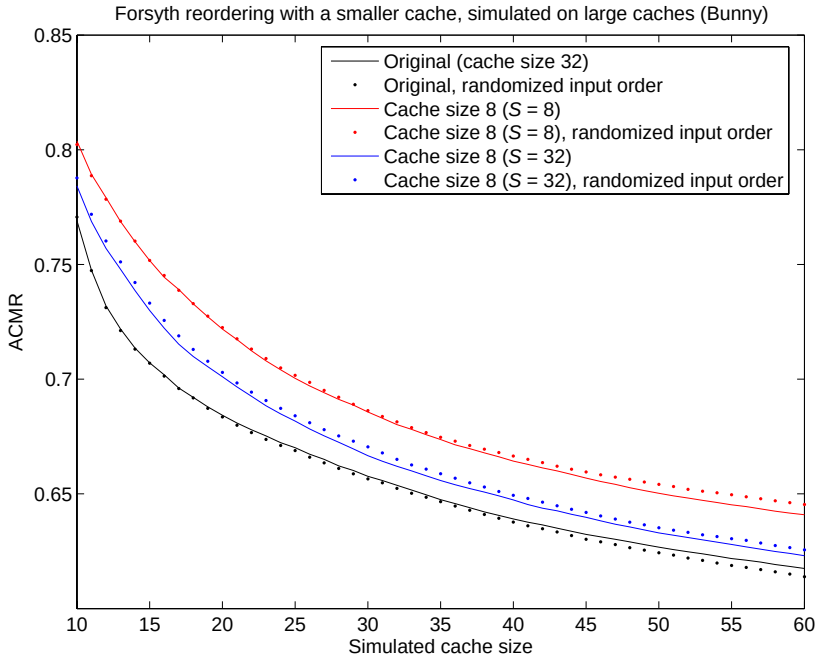

Figure 9.6: Comparison of the original cache table size of 32 elements and a cache table of 8 elements with two different scoring functions, for large cache sizes.

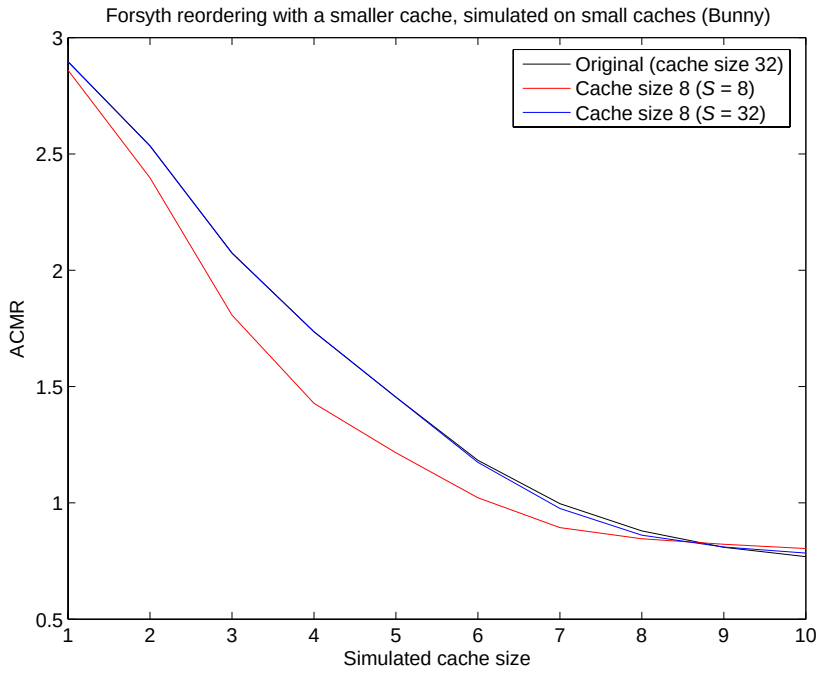

Figure 9.7: Comparison of the original cache table size of 32 elements and a cache table of 8 elements with two different scoring functions, for small cache sizes.

These assumptions are confirmed by simulations. Figure 9.6 illustrates the differences between the original cache size 32, a cache of size 8 with unmodified cache score function (approaching zero at the end of the cache) and a cache of size 8 with the same cache score function values as for a cache of size 32.

The simulations show that a cache of size 8 raises the ACMR by up to 0.05 (in the cache range 10 - 60) compared to the original cache size 32, but a cache of size 8 with the same cache score function values as the original cache only raises the ACMR by less than 0.03 (in the same range).

For simulations using caches smaller than 10, the algorithm using a smaller cache with a cache function score approaching zero at the end of the cache ($S = 8$) gave lower ACMR, as shown in figure 9.7. This score function prefers reusing vertices which have been used very recently, which is a better choice for small caches.

Chapter 10

Comparison between tipsify and Forsyth's algorithm

10.1 Similarities and differences

Even though tipsify and Forsyth's algorithm use different approaches, there are clear similarities. Both are designed to reorder triangles in a mesh in fast, linear time, which sets some limitations on the algorithms.

From a high level, the structure in both algorithms is the same. Initially, some data structures are built, for fast access later. One element (vertex or triangle) is chosen, one or a few triangles are emitted, and the following element is chosen among a small, almost fixed size, set of candidates using a simple heuristic method. If no suitable candidate is available, they continue scanning the mesh linearly for a suitable point to continue from.

Both use a simple adjacency data structure. The original index buffer given as input to the algorithm allows triangle indices to be mapped to the vertex indices for that triangle, and the adjacency data structure allows mapping in the other direction, from a vertex index to the triangles using it. Additionally, both share the trivial list of flags showing which triangles have already been added to the output.

The differences lie in the heuristic approach used. Forsyth's algorithm chooses one single triangle to emit at a time and does much work to update the data structures after adding each triangle, while tipsify emits a handful of triangles at a time for each choice made.

10.2 Simulated results

To compare the algorithms, their output orders were simulated on a range of FIFO caches. This was tested on three different models, the Stanford bunny, the dragon model and the car mesh from JBenchmark. The simulation results are depicted in figure 10.1.

None of the algorithms is the best choice in all situations. On the bunny model, tipsify gives the lowest ACMR by a relatively wide margin for the larger cache sizes, assuming that the cache size is known. On the dragon model, tipsify still achieves the lowest ACMR, but with a much smaller margin. In the car model, Forsyth's algorithm gives better results for all cache sizes. But keep in mind that tipsify gives strongly suboptimal results if the cache size is overestimated.

10.3 Algorithm run time performance

The run times for the different variations of the algorithms are listed in table 10.1. The table additionally mentions the run time for the overdraw reduction combined with tipsify. The measurements confirm the earlier discussions. Removing or limiting the size of the dead-end stack does improve the run time of tipsify a little.

The optimisation of Forsyth's algorithm by shrinking the size of the simulated cache (as described in section 9.2.1) proves to give very large performance improvements. Changing from a cache of size 32 to size 8 cuts the run time in half. Taking the marginal effect on ACMR into consideration, the optimisation is definitively recommendable in all cases where the run time is critical.

The clustering process needed for overdraw reduction turns out to be heavy. Compared to the original triangle reordering in tipsify, clustering makes the run times almost three times longer. This is probably mostly due to the floating point calculations needed for summing and averaging the cluster centre and normal.

Forsyth's algorithm also turns out to be more than three times slower than tipsify. This is due to the fact that Forsyth's algorithm chooses each triangle individually instead of emitting a few triangles for each choice made as in tipsify. Selecting each triangle individually as in Forsyth's algorithm may allow for better ACMR results, though. Forsyth's algorithm additionally needs heavy data structure update work, compared to tipsify.

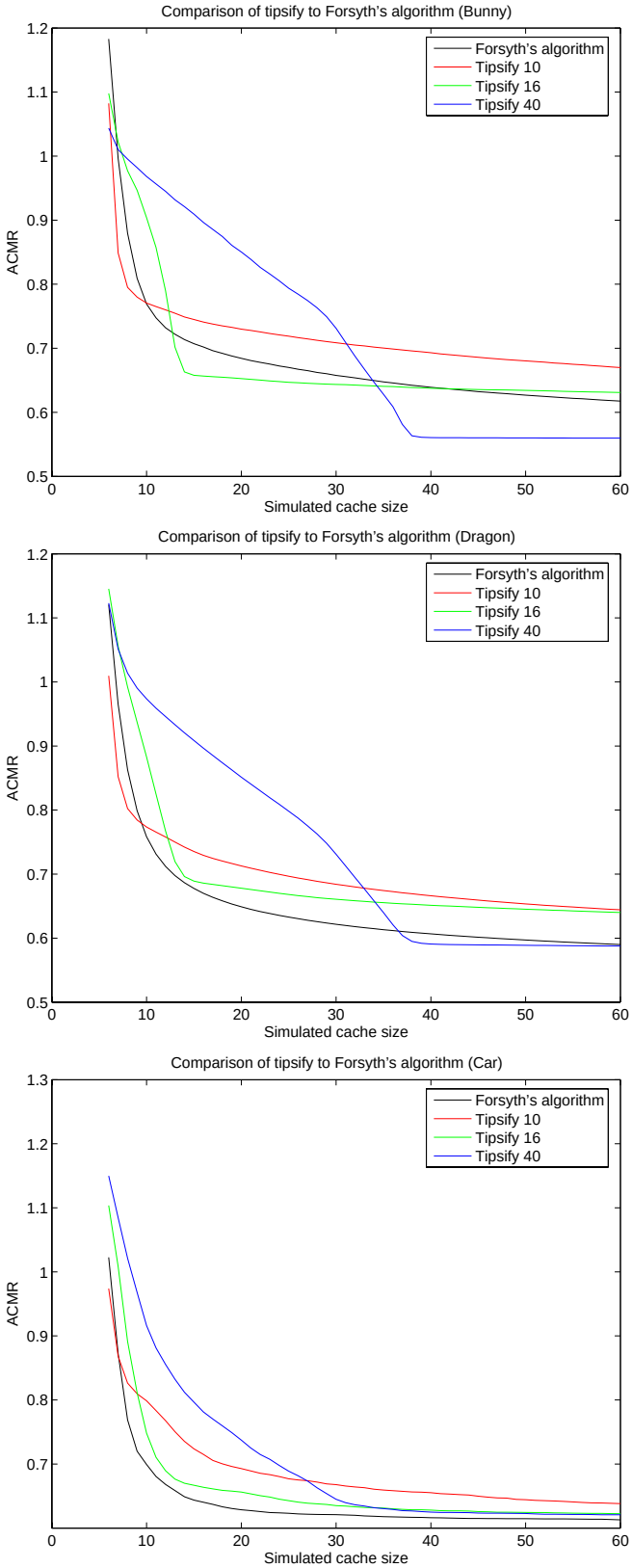

Figure 10.1: Comparison of ACMR between tipsify and Forsyth's algorithm on three different models.

	Bunny1	Bunny2	Bunny3	Bunny4
Tipsify + overdraw reduction	633	149	40.2	15.49
Original tipsify (unlimited DES)	206	45	12.5	5.78
Tipsify, DES 128	189	43	11.9	5.47
Tipsify, no DES	179	40	11.1	5.31
Forsyth's, cache 32	1233	267	58.3	13.28
Forsyth's, cache 16 (modified)	832	182	39.2	8.91
Forsyth's, cache 8 (modified)	599	130	27.2	6.41
Forsyth's, cache 8 (original)	588	125	26.6	6.09
Triangles	69451	16301	3851	948

Table 10.1: Run times (in milliseconds) for the algorithm variants, on the Stanford bunny, on an E51. Tipsify was run with different kinds of dead-end stack (DES), and the original version of tipsify was run with clustering for overdraw reduction. Forsyth's algorithm was tested with different cache sizes. Sizes 32 and 8 were tested with the original score function. Sizes 16 and 8 were also tested with the same score function values as for size 32, marked modified in the table.

The run times of both algorithms vary almost completely linearly with the number of triangles, confirming the complexity analysis in the original articles. The run time as a function of the number of triangles for some algorithm variants are illustrated in figure 10.2.

10.4 Comparison conclusions

If the cache size is completely unknown, which mostly is the case when doing the reordering off-line at the content creation stage, Forsyth's algorithm generally is a safe choice. Otherwise, tipsify might give better results if the exact cache size and behaviour is known. If the reordering run time is critical, tipsify with an underestimate of the cache size might give satisfactory results.

The memory usage of the algorithms is quite similar. By applying all the algorithm specific optimisations mentioned, both use $22.625t$ bytes, with another $10t$ bytes removable from both by limiting the number of triangles to 65535 and outputting triangle indices instead of the actual vertex indices.

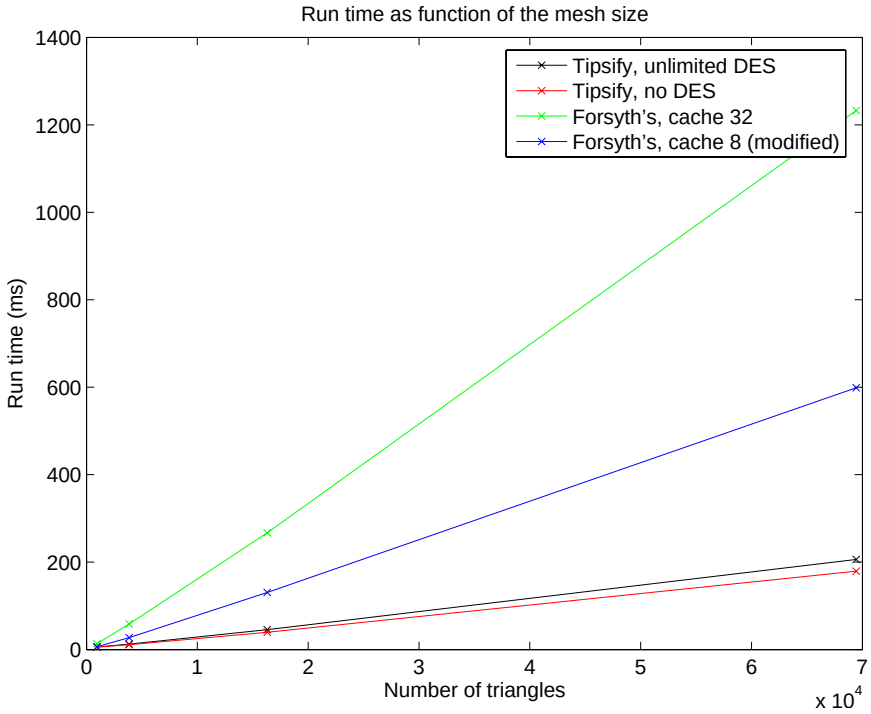

Figure 10.2: Run times as a function of the mesh size.

Chapter 11

Rendering performance tests

To evaluate the usefulness of the optimisations, frame rates for rendering different models were measured when different optimisations were applied. The results are discussed and presented below.

11.1 Triangle reordering

The maximum benefit of reordering was tested by comparing the rendering performance of a completely random triangle ordering (as the worst case), the original order and the orders generated by tipsify and Forsyth's algorithm. The orders generated by the algorithms are not the optimal orders, but they still give an estimate of the possible variation range.

The results are listed in table 11.1. These tests were run with OpenGL and OpenGL ES applications specifically constructed for these tests. The mobile results are from testing with the bunny model, the laptop results from the dragon

	E51	E65	E90	NVIDIA	ATI
Random	1.23	0.71	3.50	2.50	26.01
Original	1.68	0.94	3.50	4.28	45.42
Tipsify (best)	2.78	1.69	3.76	10.13	101.39
Forsyth's	2.67	1.64	3.67	9.95	99.97
Best tipsify parameter	38	22	6	12	15
Relative improvement	65%	80%	7%	137%	123%

Table 11.1: Comparison of frame rates for random, original and tipsified orders and orders generated by Forsyth's algorithm

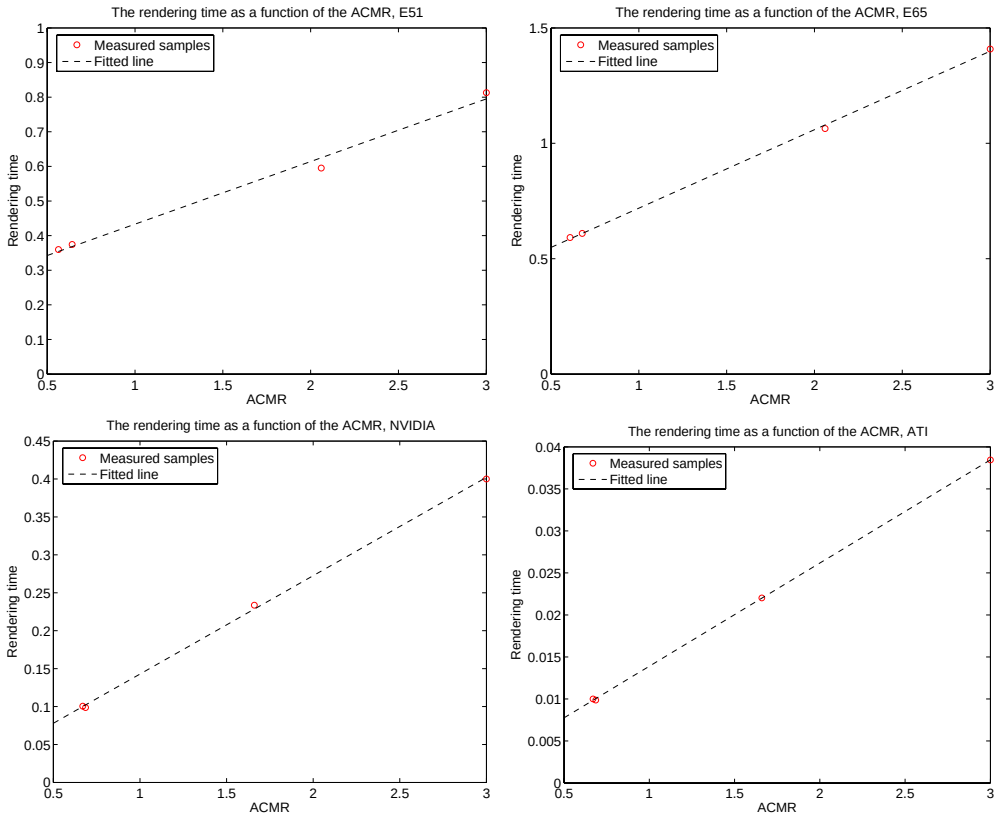

Figure 11.1: Rendering time as a function of the ACMR.

model. Note that the result from tipsify is the best achieved, even though Forsyth's algorithm might give better results than tipsify for most other parameter values to tipsify.

The actual benefit of reordering of course depends on the original ordering. For the models tested in this case, the performance of the original orderings is much closer to the performance of the random orderings than to the optimised orderings. Simulations of the meshes show an ACMR of about 2.0 for the bunny model and around 1.2 to 1.8 for dragon, depending on the simulated cache size.

Assuming that the rendering time of a frame consists of one part of constant length and one part directly proportional to the ACMR, the rendering frame rate for a theoretical ordering with an ACMR of 0.5 can be estimated. For this calculation, the actual ACMR values corresponding to the timed tests are needed. However, they are usually not available. For the random orderings, the ACMR can be assumed to be 3. The rest of the ACMR estimates can be taken from simulations. The best tipsify parameter value is used as an estimate of the cache size. The cache size for both ATI and NVIDIA is assumed to be 16, for E51 and E65 it is assumed to be 38 and 22 respectively.

	Original	Draw order	Random
bunny1-opt	2.72	2.79	2.63
bunny1	2.77	2.84	2.68
grid	9.72	9.74	9.76
hi_dog	5.66	5.74	5.58
mapanimlow	13.21	13.17	13.14
texturedcar	11.13	11.28	11.10

Table 11.2: Frame rates for different vertex orderings, on E51, with triangles reordered by tipsify

Using these four frame rate samples and their corresponding frame rates, the frame rendering time can be approximated using the method of least squares, as visualised in figure 11.1. Using the extrapolated rendering time for an ACMR of 0.5, the best achievable frame rates would be 2.91 and 1.82 for E51 and E65 respectively and 12.8 and 129.0 for NVIDIA and ATI respectively. The conclusion is that there is not much left to be gained on E51 and E65 in this area, since they have quite large caches already and the filling part of the rendering time is large. It is worth noting that the filling part of the rendering time is very large even though the test models used simple, flat shading of the triangles. ATI and NVIDIA would gain slightly more by using larger caches, allowing for even lower ACMR. The effect would in practice not be as dramatic as these tests show, though, when using more complex filling (as they are designed for) than in these test cases.

11.2 Vertex reordering

The actual gain of vertex reordering is highly dependent on the environment, for example, the hardware memory cache setup. The relative size of the gain also depends on how large the memory access time is compared to the total rendering time. To get as visible results as possible, this was tested with efficient triangle orders, instead of the original triangle orders. The tests were conducted on M3G models, with the reordering algorithms integrated into the M3G library.

The results from comparing the original vertex ordering, vertices ordered according to the draw sequence and vertices ordered randomly on an E51 are shown in table 11.2. The variation is very small, not much larger than the measurement

	Original	Draw order	Random
bunny	107.0	110.6	101.6
dragon	10.0	10.0	9.7

NVIDIA

	Original	Draw order	Random
bunny	473.1	485.5	451.5
dragon	101.3	101.7	90.1

ATI

Table 11.3: Frame rates for different vertex orderings on laptop chipsets, with triangles reordered by tipsify

noise. Results from the laptop chipsets are listed in table 11.3. The dragon model seem to have a good ordering initially, since ordering the vertices according to the drawing order gives very small improvements. The bunny model gives a little larger improvements though, but also there the difference between the best and the original case is much smaller than between the worst and the original case.

The conclusion is that the memory access part of the rendering times in the tested environments is very small. Therefore, reordering the vertices is by far not as important as reordering the triangles. If the reordering can be done without any other negative effects on, for instance, loading times (by doing the reordering at the content creation stage), it is of course recommended.

11.3 Overdraw reduction

To test the possible gain from avoiding overdraw, a special model was constructed for this situation. The model contains one single mesh, consisting of a number of concentric spheres. The best case mesh contains the triangles ordered from the outermost to the innermost (named -in), while the worst case mesh has the triangle in the opposite order, going from the innermost to the outermost (named -out). In the best case, the outer sphere is drawn initially. After that, the triangles for the rest of the spheres are rasterised, but all of them fail the depth test and do not update the colours of the pixels. This was tested for models with 3, 50 and 150 spheres. To get heavier colour calculations, the spheres are textured with trilinear filtering.

	E51		E90	
	orig.	reordered	orig.	reordered
sphere3-in	13.44	12.63	63.30	63.24
sphere3-out	9.09	12.93	63.29	63.22
sphere50-in	2.58	2.48	24.66	24.32
sphere50-out	0.78	2.45	24.46	24.31

	NVIDIA		ATI	
	orig.	reordered	orig.	reordered
sphere3-in	319.4	316.4	619.4	612.4
sphere3-out	282.7	317.4	501.5	597.4
sphere50-in	51.8	51.4	240.0	230.5
sphere50-out	32.3	51.8	80.2	234.3
sphere150-in	18.7	18.7	104.8	100.2
sphere150-out	11.9	18.4	28.4	100.0

Table 11.4: Frame rates for the test models with concentric spheres, for E51, E90 and the laptop chipsets

The test results are listed in table 11.4. On the E51, with software rendering, there is obviously a big difference for the extreme case with 50 spheres, but for the more normal case of 3 spheres and thus 3 layers of overlapping surfaces the difference is much smaller. By enabling clustering in the tipsify reordering almost similar frame rates were achieved for both kinds of each model, roughly the same frame rates as for the original best case models. The threshold λ for splitting the mesh into clusters, as described in section 5.3, was set to 0.9 in this test, to get small enough clusters.

The test results from the E90 are less obvious, though. The MBX chipset uses a technique called tile based deferred rendering, to avoid rendering pixels which will not be visible. In a sense, MBX does an overdraw reduction pass of its own, for every rendered frame. Therefore, the results are almost identical for both versions of the models.

The results from the laptop chipsets do not differ significantly in behaviour from the software rendering results. The potential gains on reasonable amounts of overdraw (the case with 3 spheres) are small.

11.3.1 Practical issues

The clustering needs to know the target threshold λ. If the mesh is large, any generic threshold (e.g., 0.7) might work fine, assuming that it gives sufficiently many small clusters. If the mesh is small and mostly limited by the fill rate, it might benefit much more from reducing overdraw than from lowering the ACMR. For small meshes, a generic λ might not be achieved at all, leaving the mesh in one single cluster in the worst case. In extreme cases, optimal overdraw reduction could be achieved by shrinking each cluster to individual triangles. Therefore, for this to be used automatically, some estimate of whether the transform rate or the fill rate is the most limiting factor is needed, guiding the choice of λ.

Another problem is that it must be confirmed that the model is intended to be viewed from the outside, before the occlusion potential estimate makes sense. Calculating the occlusion potential also needs normals, which may not be available in all meshes. (Admittedly, triangle normals can be generated, further increasing the run time of the algorithm, though.)

Most meshes occlude themselves very little. In all convex bodies, every pixel is drawn only once if backface culling is used, or at worst twice otherwise. Only the concave parts of meshes can occlude other parts of the mesh. Therefore the potential gain from enabling this on most real-world meshes is very small.

Chapter 12

M3G integration

12.1 General

The current version (1.1) of M3G limits the order of data input, both through the API and through the file format, to be in triangle strip format [AEHR05]. Generally, if possible, reordering should be done at the content creation stage, but the triangle strip format is a bottleneck when creating content for M3G. By reordering internally in M3G, better vertex cache usage is possible than what could be achieved otherwise, since the triangles can be reordered to arbitrary order without making many inefficient short strips.

On the other hand, internal reordering could spoil the initial ordering, if that ordering was more clever (e.g., by having more information about the intended usage) than what the internal reordering algorithm can achieve. Blindly optimising for reduced vertex transforming might be harmful for the performance if the mesh actually was most limited by triangle filling speed and the initial order perhaps was chosen to reduce overdraw.

The M3G specification does not explicitly state that triangles must be drawn in the order they are specified in the index buffers, giving freedom to do internal reordering within the index buffers if desired. In most cases, rendering triangles in another order does not noticeably change the end result, but if depth buffering is disabled or blending is used, the end result may actually change remarkably.

To avoid problems in these situations, even though the M3G specification would allow any drawing order, the original ordering must be preserved either explicitly as the original index buffer or implicitly as a triangle index mapping allowing the implementation to revert the index buffer to the original ordering. If a reordered

mesh is to be drawn in any of those cases, the original ordering should be used instead.

12.2 Integration

The reordering of triangles can be implemented in an M3G library completely within the TriangleStripArray class. To be able to determine the drawing mode (blending, depth testing), the Mesh class must also be modified, to pass along the Appearance object used for rendering each submesh to the IndexBuffer subclasses. If overdraw reduction is wanted, the vertex buffers are needed during the reordering. In that case, the reordering must be coordinated from the Mesh class.

The reordering must be made to fail quietly. If there is not enough memory for the temporary buffers needed to reorder the triangles, the reordering should be cancelled. This should not be reported as an error, though, since the scene is renderable equally well without reordering.

Another potential problematic situation lies in the memory usage. When adding such a non-critical enhancement as triangle reordering, scenes which fit into memory earlier must still work after adding such an enhancement, even though there might not be enough memory for the reordering for that particular scene. A reordered index buffer almost unavoidably uses more memory than the original, since the original ordering must be preserved in some way, and the reordered triangles are stored as triangle lists instead of triangle strips. In some extreme cases, though, a reordered triangle list and a mapping to restore the original ordering may even be smaller than the original triangle strips, if the original strips were very inefficient.

If each index buffer is reordered immediately when created if there is enough memory, the first index buffers will all be reordered, since there is plenty of memory at that stage. Due to the excess memory usage due to reordering of the first index buffers, there may not be enough memory to create even the absolutely essential parts of the rest of the objects. Therefore, to avoid regressions, the reordering should be done after all the essential memory allocations have been done, for example, when the first frame is rendered. In dynamic scenarios where new objects are created even after that, this solution is not sufficient.

Another slightly more complicated solution is to keep track of all the allocations

of non-essential data. When the implementation runs out of memory, it first tries to free parts of the non-essential data and then tries again.

To evaluate the benefits of integration while preserving simplicity in the test solution, reordering was implemented and integrated at the mesh creation stage.

12.2.1 Vertex reordering

Even though reordering of vertices seems like a simple operation (vertex data is reordered, and indices in the index buffers are updated according to the mapping), it is much more complicated in a general case.

Implementing it internally in M3G is not trivial. In M3G, vertex data is stored in separate buffers (separate for each vertex attribute such as position, normal, colour, texture coordinates). If vertices are reordered, the vertex data must be kept coherent by either reordering all vertex arrays associated with the same index buffers using a similar mapping or not reordering them at all. Conversely, if one vertex buffer is used by many index buffers, all index buffers must be updated with the same mapping.

Objects can be shared, meaning that the object is referenced by two or more other objects. The references are one-way, though, meaning that the objects do not know themselves which objects have references to them. If one vertex data array is shared by two different meshes, both meshes should be reordered with the same mapping. This kind of situation is illustrated in figure 12.1.

Another complicated situation is within skinned meshes. In skinned meshes, the actual vertex indices are significant and are not completely encapsulated within the class. When attaching bones to skinned meshes, the bones are attached to ranges of vertex indices. Since this, in principle, could happen at any time at run time, the mapping would have to be stored within the object, to allow conversion from the indices given as parameters to their new positions.

Given all these complications, vertex reordering internally within M3G is not recommendable.

12.3 Testing of simple static M3G files

For simple but easily measurable tests, the M3G test models mentioned in section 7.2 were used.

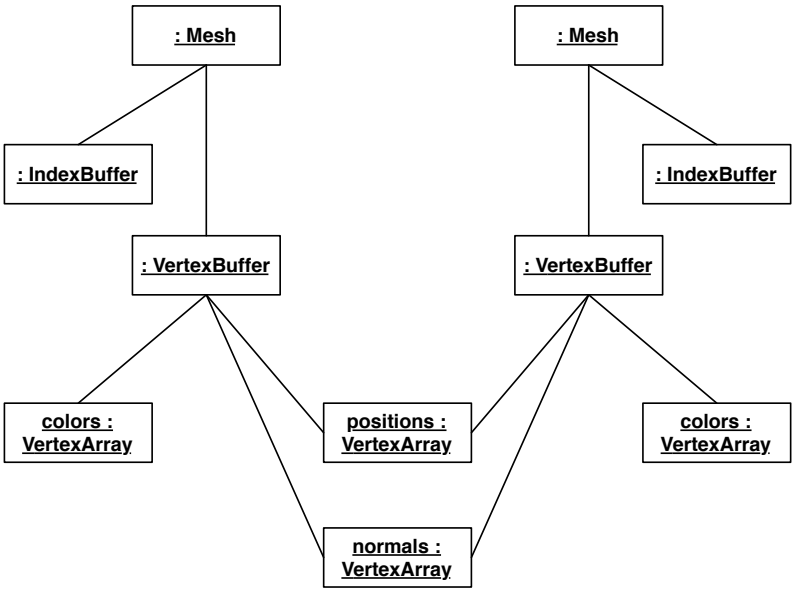

Figure 12.1: Example of shared objects in M3G.

	Original	Tipsify 30	Forsyth	Lists	Mixed
bunny1-opt	1.91	2.72	2.63	1.92	1.92
bunny1	1.56	2.78	2.68	1.66	1.67
grid	9.74	9.86	9.93	9.81	9.85
hi_dog	5.39	5.43	5.65	5.43	5.46
mapanimlow	13.19	13.48	13.35	13.13	13.30
texturedcar	10.53	11.18	11.24	10.64	10.65

Table 12.1: Benchmarks on E51, measurements given in frames per second

The models were tested by rendering them unmodified, reordered by tipsify with a target parameter of 30, reordered by Forsyth's algorithm, converted to triangle lists and selectively converted to lists if the IPT value for a submesh was over 3.

Results from testing on E51 are shown in table 12.1. The results show large gains on the bunny model, which intentionally is used as a very vertex transform intensive model. The other models consisting of actual M3G content also achieve speedups, but much smaller gains, since they have fewer vertices, less complex vertex transforming (e.g., usually no lighting at all) and more heavy pixel filling (using texturing).

On E65, the tests mostly showed similar improvements as on E51, as shown in table 12.2. The relative improvements were of the same magnitude as on E51.

	Original	Tipsify 30	Forsyth	Lists	Mixed
bunny1-opt	1.21	1.57	1.57	1.15	1.12
bunny1	0.82	1.57	1.57	0.89	0.89
grid	6.57	6.65	6.64	6.61	6.53
hi_dog	3.70	3.83	3.84	3.68	3.68
mapanimlow	7.89	8.13	8.13	7.96	7.89
texturedcar	6.34	6.80	6.88	6.45	6.44

Table 12.2: Benchmarks on E65, measurements given in frames per second

	Original	Tipsify 30	Forsyth	Lists	Mixed
bunny1-opt	5.41	4.00	3.54	7.04	5.41
bunny1	1.76	3.50	3.50	3.30	3.32
grid	21.33	21.06	20.23	21.13	21.50
hi_dog	8.94	9.57	9.36	11.61	9.75
mapanimlow	31.61	31.43	29.89	31.19	32.01
texturedcar	51.50	40.84	39.91	46.13	51.88

Table 12.3: Benchmarks on E90, measurements given in frames per second

Benchmark results from the E90 are listed in table 12.3. Most models except hi_dog and bunny1 show no improvements when reordering for vertex cache utilisation (tipsify 30 and Forsyth's algorithm) and actually perform worse. Hi_dog and bunny1-opt perform better than both the original version and the reordered versions when preserving the original ordering and just converting the triangle strips into lists. These results, and the earlier conclusion that the cache is very small, matches the hints in Best Practices for HW-Accelerated Graphics Optimization from Forum Nokia [FN07], which says that the optimal format for the MBX chip is strip-ordered triangles in triangle lists.

The mixed setting (converting submeshes from strips to lists if their IPT value is over 3) gives better results than converting everything to lists for some meshes, but worse results for some. By adjusting the threshold for conversion, those results can be improved further, but no single threshold value gives the best results for all meshes.

12.4 Testing of ordinary applications

When testing the effect on triangle reordering on ordinary applications, the original orderings were compared to the orderings returned by tipsify with the parameter 30, on an E51.

Testing results from testing the applications JBenchmark HD, JBenchmark PRO and Ducati 3D Extreme are listed in tables 12.4, 12.5 and 12.6 respectively. The measurements without an unit specified are unitless scores.

In most test cases, the results from triangle orders reordered by tipsify were as good as or better than the original results. The largest improvements were observed in the synthetic test cases. In some of the scenes simulating complete games, tipsify actually made the results a little worse. The loading performance as tested by JBenchmark PRO was, naturally, worse when doing the triangle reordering.

	Original	Tipsify	Unit
Smooth triangles	105330	111287	Triangles/s
Textured triangles	84535	87853	Triangles/s
Gaming	162	159	
Gaming	5.4	5.3	Frames/s

Table 12.4: JBenchmark HD test results

	Original	Tipsify	Unit
Lights: Ambient x 1	120548	132808	Triangles/s
Lights: Omni x 1	114419	125656	Triangles/s
Lights: Parallel x 1	138937	150175	Triangles/s
Lights: Parallel x 2	135872	147110	Triangles/s
Lights: Parallel x 4	128721	139959	Triangles/s
Lights: Parallel x 8	119527	130764	Triangles/s
Lights: Spot x 1	119527	131786	Triangles/s
M3G CarRace - LQ	727	735	Frames
M3G CarRace - MQ	381	397	Frames
M3G FPS - LQ	288	295	Frames
M3G FPS - MQ	193	197	Frames
M3G loading	12	10	
MorphingMesh	209	216	Frames
SkinnedMesh: Few Bones	307	320	Frames
SkinnedMesh: Many Bones	273	292	Frames
Triangles: Flat Shaded + Color	162434	173672	Triangles/s
Triangles: Smooth Shaded + Color	150175	159369	Triangles/s
Triangles: Textured	127700	134851	Triangles/s
Triangles: Textured + Color	118505	123613	Triangles/s

Table 12.5: JBenchmark PRO test results. Higher numbers are better in all of the cases.

	Original	Tipsify	Unit
Score	40.2	41.9	
FPS	6.9	7.2	Frames/s

Table 12.6: Ducati 3D Extreme test results

Chapter 13

Conclusions

13.1 Summary of results

The presented improvements to the algorithms make them use less memory or give them better run time performance, making them more suitable for use in embedded environments. Limiting the dead-end stack in tipsify to a fixed size reduces the algorithm's peak memory consumption by approximately 30%, and by simulating a smaller cache in Forsyth's algorithm, the run time of that algorithm is cut in half.

Their memory usage is low enough and they run fast enough to be used on the target devices. The goal that was defined for the algorithm improvements can thus be considered achieved. Tipsify has a definitive advantage over Forsyth's algorithm regarding run time, but if the cache size is unknown Forsyth's algorithm may be a better choice.

The vertex cache utilisation algorithms give very large improvements in specific test cases where vertex transforming is the largest bottleneck, for example, improving the frame rate by over 60%. Most of the tested existing M3G content, however, is mainly limited by the fill rate of the renderer. Thus, vertex cache optimisations generally only give minor improvements on the existing M3G content, especially on software renderers where the fill rate generally is a bottleneck.

Vertex reordering gives only minor improvements, and overdraw reduction only improves very specific cases.

Converting triangle strips to triangle lists if the strips are inefficient is a simple optimisation that saves memory, and can be enabled without compromising on any other factor. In some specific cases such as on the MBX chip on E90, this

also gives rendering time improvements, since the chip prefers the triangle list format.

As for integrating the optimisations into scene graph frameworks, vertex reordering was determined to be too complicated when vertex and triangle data is spread across sharable objects. Overdraw reduction cannot be enabled automatically, since it assumes knowledge about the meshes which is not easily available. Converting inefficient triangle strips to lists has no direct drawbacks and can be recommended generally.

Triangle reordering for vertex cache utilisation can easily be integrated. In most cases, it either improves the performance or does not affect it at all. Only in a few cases did it degrade the performance a little. The benefit on the tested existing M3G content was small in general, though. Therefore, considering that it requires more memory both during the reordering and afterwards when storing the original order, the benefits are not large and clear enough to outweigh the possible risks. Enabling this cannot be recommended unanimously, but may still be considered.

13.2 Future work

The embedded environments will keep on developing and they will get even better graphics capabilities and less limiting processing and memory resources. The need for optimisations will not decrease, though, if products are to stay competitive. The current, unoptimised 3D content will still be available. Instead of updating and optimising all of the old content, methods of automatically optimising any content will become even more desired.

The current algorithms for reordering triangles for better vertex cache usage give very good results. The main target in improving the algorithms is not getting even better cache utilisation, since the current ones already are quite close to the theoretical limits. Instead, the memory usage and run time could still be improved even if it would compromise on the ACMR results. An algorithm combining the properties of tipsify and Forsyth's algorithm would be desirable, running as fast as tipsify but generating orders universally efficient on any cache size without targeting any specific size.

If these kinds of optimisations are to be enabled automatically and transparently, methods for fast detection of the bottlenecks are needed. If the view independent overdraw reduction is to be used, the properties and usage of the model must be

detected and a suitable value for the clustering threshold λ has to be estimated. If a model is well optimised initially, that fact should be detected, to avoid unnecessary optimisation. The cases where the current reordering algorithms degrade the performance should be detected and analysed, in order to either improve the algorithms or avoid doing reordering in those cases.

Sammanfattning

Introduktion

Inom realtidsrenderad 3D-grafik i mobila och övriga inbyggda datorsystem är man ofta tvungen att göra stora kompromisser för att nå acceptabel prestanda. De här kompromisserna försämrar ofta den visuella kvaliteten på renderingen. Realtidsrenderade 3D-modeller är vanligtvis konstruerade av trianglar. Det finns dock en grupp optimeringar som bygger på att ändra ordning eller format på triangeldatan, vilket inte förändrar kvaliteten på slutresultatet men kan ge markant bättre prestanda i vissa fall.

Dessa optimeringar kunde till viss del appliceras på 3D-modellerna då de skapas, men genom att integrera dem i biblioteksrutiner kan man få allt innehåll att dra nytta av optimeringarna utan att explicit uppdatera och optimera alla existerande modeller.

Optimeringar baserade på ordning och format

Cachning av transformerade noder

Då trianglar renderas så måste de noder (eng. vertices) som används transformeras från de koordinater de har inom objektet till koordinater i världens koordinatsystem och därifrån vidare till kamerans koordinatsystem, för att till slut projiceras till 2D-koordinater. Samtidigt transformeras normalvektorerna för att beräkna ljussättningskoefficienter. De här transformationerna är tunga operationer och man vill således göra så få transformationer som möjligt. Genom att mellanlagra resultatet av transformationen kan man reducera antalet transformationer som behöver göras.

Tidigare utforskades idéer om att explicit hantera en buffert för dessa transformerade noder. Bland andra Bar-Yehuda och Gotsman [BYG96] presenterade

metoder för hur detta kunde göras. De här metoderna slog dock inte igenom i praktiken. Hoppe presenterade dock 1999 idén att problemet kunde lösas med en transparent cache [Hop99]. På så sätt krävs inga förändringar i de gränssnitt eller dataformat som används.

För att effektivt utnyttja en eventuell cache för transformerade noder bör man ordna om trianglarna så att de används i en sådan ordning att trianglarna återanvänder noder som nyligen använts. Många algoritmer har utvecklats för att lösa problemet att ordna trianglarna på ett sådant sätt, men de flesta algoritmerna har varit relativt långsamma.

Forsyth presenterade 2006 en algoritm som ordnar om trianglar för att ge ordningar som är effektiva oberoende av den exakta storleken på cachen [For06]. Det anmärkningsvärda med den här algoritmen är att den kör i snabb linjär tid. Sander m.fl. introducerade 2007 en motsvarande algoritm kallad tipsify [SNB07], som även den kör i linjär tid. Till skillnad från Forsyths algoritm så försöker tipsify optimera ordningen för en given cachestorlek.

Sortering av noddata

Då en nod transformeras måste datan för den noden läsas från minnet från den buffert där datan finns lagrad. Läsningen är effektivare om den sker så sekventiellt som möjligt, dvs. data läses från minnesplatser nära andra platser som nyligen lästs. För att åstadkomma en nästintill sekventiell läsordning kan man ordna om noddatan genom att skapa en ny tom buffert för noddata och börja gå igenom triangeldatan. Då en nod används som inte använts tidigare sätts den till efter den senast tillsatta noden i den nya bufferten. Indexen i triangeldatan uppdateras samtidigt enligt vart noden flyttats i den nya bufferten. På så sätt flyttar man bara om noderna inom bufferten, trianglarna ritas fortfarande i samma ordning som tidigare.

Med en sådan ordning kommer alla noder att läsas sekventiellt, förutom de noder som återanvänds och som inte finns mellanlagrade i en cache för transformerade noder.

Minskning av överritning

Då flera trianglar överlappar samma bildpunkter, används ofta tekniken z-buffring för att avgöra vilken av trianglarna som ska synas i vilken bildpunkt. En buffert

innehåller djupkoordinaten för varje ritad bildpunkt. Då en ny triangel ritas kan man med hjälp av de här djupkoordinaterna avgöra huruvida färgen på varje bildpunkt ska uppdateras. Om beräkningen av färgen är tung, t.ex. om stora texturer används tillsammans med avancerad filtrering, kan man spara mycket arbete genom att rita om bildpunkterna så få gånger som möjligt.

Således kan man spara arbete genom att först rita de trianglar som kommer att vara synliga, för att senare rita de trianglar som kommer att vara mer skymda. På så sätt uppdateras färgen för varje bildpunkt färre gånger och färre färgberäkningar görs jämfört med om trianglarna ritas i godtycklig ordning.

Vilka trianglar som är synliga och vilka som är skymda beror naturligtvis på varifrån scenen betraktas. Sander m.fl. presenterade dock en metod för att sortera kluster av trianglar för att minska överritning oberoende av betraktningsvinkel [SNB07]. Genom att sortera hela kluster istället för individuella trianglar kan man bibehålla effektiv användning av en nodcache inom klustren, förutsatt att klustren är tillräckligt stora. Då kan också klustren ritas i vilken ordning som helst, utan att försämra nodcache-prestandan.

Konvertering av triangeldata

Triangeldata lagras vanligtvis antingen som triangellistor, där alla tre noder i triangeln är explicit angivna, eller som triangelremsor (eng. triangle strips), där en triangel definieras av de två senaste noderna samt en ny. Om remsorna är effektiva tar de mindre lagringsutrymme än fullständiga listor. Att konstruera effektiva remsor är dock inte helt trivialt och det förekommer i praktiken en hel del ineffektiva remsor.

Om data anges som remsor och det framkommer att remsorna är ineffektiva kan man således spara minne och därmed också minnesbandbredd genom att konvertera dem till triangellistor istället.

Förbättringar och kompromisser

Minnesanvändning i tipsify

Algoritmen tipsify innehåller en stack där tidigare använda noder lagras. Den här stacken används för att hitta nya närbelägna punkter att fortsätta ifrån då

algoritmen gått in i en återvändsgränd, där alla närbelägna trianglar redan har behandlats.

Det är dock svårt att uppskatta hur stor stacken kommer att bli, så om man vill undvika att allokera mera minne åt stacken under algoritmens gång är man tvungen att bestämma en övre gräns för dess storlek. I praktiken används högst ungefär hälften av det maximala antalet element.

Simuleringar visar att slutresultatet endast blir marginellt sämre om man lämnar bort den här stacken helt. Ännu bättre kompromisser nås om man använder en stack av en begränsad storlek, där endast de senaste N elementen lagras. Med storleken $N = 128$ är cacheanvändningen nästan lika effektiv som då stacken var obegränsad, trots att man sparar in mycket på minnesanvändningen.

Uppskattning av antalet noder att transformera

Ett av stegen i tipsify går ut på att uppskatta hur många nya noder som måste transformeras för att rita ett visst antal nya trianglar. Den ursprungliga algoritmen använde för säkerhets skull en överskattning av antalet noder som måste transformeras. I vissa fall åstadkommer den här överskattningen dock kraftigt suboptimala resultat. Det visar sig att mer realistiska uppskattningar ger bättre resultat i dessa samt några andra fall.

Effektivering av Forsyths algoritm

Forsyths algoritm simulerar en cache av en viss storlek, trots att den genererar ordningar som ska fungera bra oavsett hur stor den riktiga cachen är. Algoritmens körtid beror till stor del på hur stor cache som simuleras. Genom att krympa den simulerade cachen får man algoritmen att köra mycket snabbare, men man offrar samtidigt en del av slutresultatet. Om man därtill justerar poängsättningsfunktionen som algoritmen använder blir den negativa inverkan på slutresultatet mycket mindre.

Jämförelse av algoritmerna

Tipsify ger oftast bättre resultat än Forsyths algoritm, förutsatt att man känner till hur stor cachen är. Om man inte känner till det och överskattar cachens storlek ger tipsify mycket suboptimala resultat. Därför är Forsyths algoritm ett säkrare

val om man inte känner till cachens storlek eller inte har möjlighet att analysera dess beteende genom att testa med olika parametrar till tipsify. Tipsify är dock mer än tre gånger snabbare än Forsyths algoritm på att sortera trianglar.

Testresultat

Algoritmerna för att sortera trianglarna för att förbättra användningen av nodcachen ger stora förbättringar av renderingstiderna på modeller där transformeringen av noder är den största flaskhalsen. En testmodell som ursprungligen renderades med 1,68 bilder per sekund på en telefon renderades med 2,78 bilder per sekund då den optimerats. Modeller som förekommer i riktiga tillämpningar konstruerade med mobiltelefoner i åtanke förbättras inte lika mycket, eftersom flaskhalsen i sådana modeller främst är att rasterisera och fylla trianglarna. För sådana modeller förbättras renderingshastigheten endast med några procent.

Att sortera noddatan för att läsa data i en mer sekventiell ordning ger ännu mindre förbättringar, förbättringarna rör sig om någon enstaka procent.

Om trianglarna sorteras för att minska överritning kan man vinna väldigt mycket, förutsatt att modellen i fråga faktiskt överlappar sig själv i många lager. En fullständigt konvex modell överlappar aldrig sig själv och för övriga modeller är det endast de konkava partierna som eventuellt kan överlappa andra delar av modellen. Därför är det tveksamt hur stor nytta den här optimeringen gör i praktiken.

Resultatanalys

Algoritmerna är tillräckligt snabba och minnessnåla för att kunna användas integrerade i bibliotek i inbyggda miljöer, framför allt om de förbättringar som presenterats används. Nyttan av att sortera trianglarna för förbättrad användning av nodcachen är dock inte tillräckligt stor på största delen av de 3D-modeller som förekommer för att uppväga riskerna i att sorteringen kräver en del minne.

Sortering av noddata är för komplicerat att implementera i generella bibliotek där noddatan är utspridd i många objekt som kan användas i flera modeller inom samma scen.

Sanders metod för att sortera triangelkluster för att undvika överritning går inte

att tillämpa automatiskt på alla modeller eftersom den kräver information om hur modellerna kommer att användas, vilket inte finns tillgängligt automatiskt.

Att konvertera ineffektiva triangelremsor till listor är dock en enkel och säker optimering som sparar minne och kan appliceras automatiskt utan risk.

Bibliography

[AEHR05] Tomi Aarnio, Sean Ellis, Jyri Huopaniemi, and Kimmo Roimela. *JSR 184: Mobile 3D Graphics API for J2ME^{TM}*. Java Community Process, August 2005.

[BG02] Alexander Bogomjakov and Craig Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum*, 21(2):137–148, 2002.

[BM07] David Blythe and Aaftab Munshi. *OpenGL® ES Common/Common-Lite Profile Specification*. The Khronos Group, April 2007. Version 1.1.10 (Full Specification).

[BYG96] Reuven Bar-Yehuda and Craig Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Trans. Graph.*, 15(2):141–152, 1996.

[FN07] Forum Nokia. *Best Practices for HW-Accelerated Graphics Optimization*, October 2007.

[For06] Tom Forsyth. Linear-speed vertex cache optimisation. `http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html`, September 2006.

[Hop99] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[LY06] Gang Lin and Thomas P. Y. Yu. An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006.

[SAF+06] Mark Segal, Kurt Akeley, Chris Frazier, John Leech, and Pat Brown. *The OpenGL® Graphics System: A Specification*. The OpenGL Architecture Review Board, December 2006. Version 2.1.

[SNB07] Pedro V. Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 89, New York, NY, USA, 2007. ACM Press.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 2*. Addison-Wesley Professional, fifth edition, August 2005.

[vKdSP04] Oliver Matias van Kaick, Murilo Vincente Gonçalves da Silva, and Hélio Pedrini. Efficient generation of triangle strips from triangulated meshes. *Journal of WSCG*, 12(1-3):475–481, February 2004.

Appendix A

Implementation of the tipsify algorithm

Listing A.1: Sample implementation of Tipsify

```
1  #define __STDC_LIMIT_MACROS
2  #include <stdint.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define DEAD_END_STACK_SIZE 128
7  #define DEAD_END_STACK_MASK (DEAD_END_STACK_SIZE - 1)
8
9
10 // The size of these data types control the memory usage
11 typedef uint8_t AdjacencyType;
12 #define MAX_ADJACENCY UINT8_MAX
13
14 typedef int32_t VertexIndexType;
15 typedef int32_t TriangleIndexType;
16 typedef int32_t ArrayIndexType;
17
18 #define ISEMITTED(x)  (emitted[(x) >> 3] &  (1 << (x & 7)))
19 #define SETEMITTED(x) (emitted[(x) >> 3] |= (1 << (x & 7)))
20
21 // Find the next non-local vertex to continue from
22 int skipDeadEnd(const AdjacencyType* liveTriangles,
23                 const VertexIndexType* deadEndStack,
24                 int& deadEndStackPos,
25                 int& deadEndStackStart,
26                 int nVertices,
27                 int& i) {
```

```
28
29      // Next in dead-end stack
30      while ((deadEndStackPos & DEAD_END_STACK_MASK) !=
            deadEndStackStart) {
31          int d = deadEndStack[(--deadEndStackPos) &
                DEAD_END_STACK_MASK];
32          // Check for live triangles
33          if (liveTriangles[d] > 0)
34              return d;
35      }
36      // Next in input order
37      while (i + 1 < nVertices) {
38          // Cursor sweeps list only once
39          i++;
40          // Check for live triangles
41          if (liveTriangles[i] > 0)
42              return i;
43      }
44      // We are done!
45      return -1;
46  }
47
48  // Find the next vertex to continue from
49  int getNextVertex(int nVertices,
50                    int& i,
51                    int k,
52                    const VertexIndexType* nextCandidates,
53                    int numNextCandidates,
54                    const ArrayIndexType* cacheTime,
55                    int s,
56                    const AdjacencyType* liveTriangles,
57                    const VertexIndexType* deadEndStack,
58                    int& deadEndStackPos,
59                    int& deadEndStackStart) {
60
61      // Best candidate
62      int n = -1;
63      // and priority
64      int m = -1;
65      for (int j = 0; j < numNextCandidates; j++) {
66          int v = nextCandidates[j];
67          // Must have live triangles
68          if (liveTriangles[v] > 0) {
69              // Initial priority
70              int p = 0;
```

```
71              // In cache even after fanning?
72              if (s - cacheTime[v] + 2*liveTriangles[v] <= k)
73                  // Priority is position in cache
74                  p = s - cacheTime[v];
75              // Keep best candidate
76              if (p > m) {
77                  m = p;
78                  n = v;
79              }
80          }
81      }
82      // Reached a dead-end?
83      if (n == -1) {
84          // Get non-local vertex
85          n = skipDeadEnd(liveTriangles, deadEndStack,
86                          deadEndStackPos, deadEndStackStart,
87                          nVertices, i);
88      }
89      return n;
90  }
91
92  // The main reordering function
93  VertexIndexType* tipsify(const VertexIndexType* indices,
94                           int nTriangles,
95                           int nVertices,
96                           int k) {
97      // Vertex-triangle adjacency
98
99      // Count the occurrances of each vertex
100     AdjacencyType* numOccurrances = new AdjacencyType[nVertices];
101     memset(numOccurrances, 0, sizeof(AdjacencyType)*nVertices);
102     for (int i = 0; i < 3*nTriangles; i++) {
103         int v = indices[i];
104         if (numOccurrances[v] == MAX_ADJACENCY) {
105             // Unsupported mesh,
106             // vertex shared by too many triangles
107             delete [] numOccurrances;
108             return NULL;
109         }
110         numOccurrances[v]++;
111     }
112
113     // Find the offsets into the adjacency array for each vertex
114     int sum = 0;
115     ArrayIndexType* offsets = new ArrayIndexType[nVertices+1];
```

```
116     int maxAdjacency = 0;
117     for (int i = 0; i < nVertices; i++) {
118         offsets[i] = sum;
119         sum += numOccurrances[i];
120         if (numOccurrances[i] > maxAdjacency)
121             maxAdjacency = numOccurrances[i];
122         numOccurrances[i] = 0;
123     }
124     offsets[nVertices] = sum;
125
126     // Add the triangle indices to the vertices it refers to
127     TriangleIndexType* adjacency = new TriangleIndexType[3*
            nTriangles];
128     for (int i = 0; i < nTriangles; i++) {
129         const VertexIndexType* vptr = &indices[3*i];
130         adjacency[offsets[vptr[0]] + numOccurrances[vptr[0]]] = i;
131         numOccurrances[vptr[0]]++;
132         adjacency[offsets[vptr[1]] + numOccurrances[vptr[1]]] = i;
133         numOccurrances[vptr[1]]++;
134         adjacency[offsets[vptr[2]] + numOccurrances[vptr[2]]] = i;
135         numOccurrances[vptr[2]]++;
136     }
137
138     // Per-vertex live triangle counts
139     AdjacencyType* liveTriangles = numOccurrances;
140
141     // Per-vertex caching time stamps
142     ArrayIndexType* cacheTime = new ArrayIndexType[nVertices];
143     memset(cacheTime, 0, sizeof(ArrayIndexType)*nVertices);
144
145     // Dead-end vertex stack
146     VertexIndexType* deadEndStack = new VertexIndexType[
            DEAD_END_STACK_SIZE];
147     memset(deadEndStack, 0, sizeof(VertexIndexType)*
            DEAD_END_STACK_SIZE);
148     int deadEndStackPos = 0;
149     int deadEndStackStart = 0;
150
151     // Per triangle emitted flag
152     uint8_t* emitted = new uint8_t[(nTriangles + 7)/8];
153     memset(emitted, 0, sizeof(uint8_t)*((nTriangles + 7)/8));
154
155     // Empty output buffer
156     TriangleIndexType* outputTriangles = new TriangleIndexType[
            nTriangles];
```

```
157     int outputPos = 0;
158
159     // Arbitrary starting vertex
160     int f = 0;
161     // Time stamp and cursor
162     int s = k + 1;
163     int i = 0;
164
165     VertexIndexType* nextCandidates = new VertexIndexType[3*
            maxAdjacency];
166
167     // For all valid fanning vertices
168     while (f >= 0) {
169         // 1-ring of next candidates
170         int numNextCandidates = 0;
171         int startOffset = offsets[f];
172         int endOffset = offsets[f+1];
173         for (int offset = startOffset; offset < endOffset; offset++)
                {
174             int t = adjacency[offset];
175             if (!ISEMITTED(t)) {
176                 const VertexIndexType* vptr = &indices[3*t];
177                 // Output triangle
178                 outputTriangles[outputPos++] = t;
179                 for (int j = 0; j < 3; j++) {
180                     int v = vptr[j];
181                     // Add to dead-end stack
182                     deadEndStack[(deadEndStackPos++) &
                            DEAD_END_STACK_MASK] = v;
183                     if ((deadEndStackPos & DEAD_END_STACK_MASK) ==
184                         (deadEndStackStart & DEAD_END_STACK_MASK))
185                         deadEndStackStart = (deadEndStackStart + 1)
                                & DEAD_END_STACK_MASK;
186                     // Register as candidate
187                     nextCandidates[numNextCandidates++] = v;
188                     // Decrease live triangle count
189                     liveTriangles[v]--;
190                     // If not in cache
191                     if (s - cacheTime[v] > k) {
192                         // Set time stamp
193                         cacheTime[v] = s;
194                         // Increment time stamp
195                         s++;
196                     }
197                 }
```

```
198                 // Flag triangle as emitted
199                 SETEMITTED(t);
200             }
201         }
202         // Select next fanning vertex
203         f = getNextVertex(nVertices, i, k, nextCandidates,
204                           numNextCandidates, cacheTime, s,
205                           liveTriangles, deadEndStack,
206                           deadEndStackPos, deadEndStackStart);
207     }
208
209     // Clean up
210     delete [] nextCandidates;
211     delete [] emitted;
212     delete [] deadEndStack;
213     delete [] cacheTime;
214     delete [] adjacency;
215     delete [] offsets;
216     delete [] numOccurrances;
217
218     // Convert the triangle index array into a full triangle list
219     VertexIndexType* outputIndices = new VertexIndexType[3*
            nTriangles];
220     outputPos = 0;
221     for (int i = 0; i < nTriangles; i++) {
222         int t = outputTriangles[i];
223         for (int j = 0; j < 3; j++) {
224             int v = indices[3*t + j];
225             outputIndices[outputPos++] = v;
226         }
227     }
228     delete [] outputTriangles;
229
230     return outputIndices;
231 }
```

Appendix B

Implementation of Forsyth's algorithm

Listing B.1: Sample implementation of Forsyth's algorithm

```
1  #define __STDC_LIMIT_MACROS
2  #include <stdint.h>
3  #include <math.h>
4  #include <string.h>
5
6  // Set these to adjust the performance and result quality
7  #define VERTEX_CACHE_SIZE 8
8  #define CACHE_FUNCTION_LENGTH 32
9
10 // The size of these data types affect the memory usage
11 typedef uint16_t ScoreType;
12 #define SCORE_SCALING 7281
13
14 typedef uint8_t AdjacencyType;
15 #define MAX_ADJACENCY UINT8_MAX
16
17 typedef int32_t VertexIndexType;
18 typedef int8_t  CachePosType;
19 typedef int32_t TriangleIndexType;
20 typedef int32_t ArrayIndexType;
21
22 // The size of the precalculated tables
23 #define CACHE_SCORE_TABLE_SIZE 32
24 #define VALENCE_SCORE_TABLE_SIZE 32
25 #if CACHE_SCORE_TABLE_SIZE < VERTEX_CACHE_SIZE
26 #error Vertex score table too small
27 #endif
```

```
28
29 // Precalculated tables
30 static ScoreType cachePositionScore[CACHE_SCORE_TABLE_SIZE];
31 static ScoreType valenceScore[VALENCE_SCORE_TABLE_SIZE];
32
33 #define ISADDED(x)  (triangleAdded[(x) >> 3] &  (1 << (x & 7)))
34 #define SETADDED(x) (triangleAdded[(x) >> 3] |= (1 << (x & 7)))
35
36 // Score function constants
37 #define CACHE_DECAY_POWER   1.5
38 #define LAST_TRI_SCORE      0.75
39 #define VALENCE_BOOST_SCALE 2.0
40 #define VALENCE_BOOST_POWER 0.5
41
42 // Precalculate the tables
43 void initForsyth() {
44     for (int i = 0; i < CACHE_SCORE_TABLE_SIZE; i++) {
45         float score = 0;
46         if (i < 3) {
47             // This vertex was used in the last triangle,
48             // so it has a fixed score, which ever of the three
49             // it's in. Otherwise, you can get very different
50             // answers depending on whether you add
51             // the triangle 1,2,3 or 3,1,2 - which is silly
52             score = LAST_TRI_SCORE;
53         } else {
54             // Points for being high in the cache.
55             const float scaler = 1.0f / (CACHE_FUNCTION_LENGTH - 3);
56             score = 1.0f - (i - 3) * scaler;
57             score = powf(score, CACHE_DECAY_POWER);
58         }
59         cachePositionScore[i] = (ScoreType) (SCORE_SCALING * score);
60     }
61
62     for (int i = 1; i < VALENCE_SCORE_TABLE_SIZE; i++) {
63         // Bonus points for having a low number of tris still to
64         // use the vert, so we get rid of lone verts quickly
65         float valenceBoost = powf(i, -VALENCE_BOOST_POWER);
66         float score = VALENCE_BOOST_SCALE * valenceBoost;
67         valenceScore[i] = (ScoreType) (SCORE_SCALING * score);
68     }
69 }
70
71 // Calculate the score for a vertex
72 ScoreType findVertexScore(int numActiveTris,
```

```
73                              int cachePosition) {
74      if (numActiveTris == 0) {
75          // No triangles need this vertex!
76          return 0;
77      }
78
79      ScoreType score = 0;
80      if (cachePosition < 0) {
81          // Vertex is not in LRU cache - no score
82      } else {
83          score = cachePositionScore[cachePosition];
84      }
85
86      if (numActiveTris < VALENCE_SCORE_TABLE_SIZE)
87          score += valenceScore[numActiveTris];
88      return score;
89  }
90
91  // The main reordering function
92  VertexIndexType* reorderForsyth(const VertexIndexType* indices,
93                                  int nTriangles,
94                                  int nVertices) {
95
96      // The tables need not be inited every time this function
97      // is used. Either call initForsyth from the calling process,
98      // or just replace the score tables with precalculated values.
99      initForsyth();
100
101     AdjacencyType* numActiveTris = new AdjacencyType[nVertices];
102     memset(numActiveTris, 0, sizeof(AdjacencyType)*nVertices);
103
104     // First scan over the vertex data, count the total number of
105     // occurrances of each vertex
106     for (int i = 0; i < 3*nTriangles; i++) {
107         if (numActiveTris[indices[i]] == MAX_ADJACENCY) {
108             // Unsupported mesh,
109             // vertex shared by too many triangles
110             delete [] numActiveTris;
111             return NULL;
112         }
113         numActiveTris[indices[i]]++;
114     }
115
116     // Allocate the rest of the arrays
117     ArrayIndexType* offsets = new ArrayIndexType[nVertices];
```

```
118     ScoreType* lastScore = new ScoreType[nVertices];
119     CachePosType* cacheTag = new CachePosType[nVertices];
120
121     uint8_t* triangleAdded = new uint8_t[(nTriangles + 7)/8];
122     ScoreType* triangleScore = new ScoreType[nTriangles];
123     TriangleIndexType* triangleIndices = new TriangleIndexType[3*
            nTriangles];
124     memset(triangleAdded, 0, sizeof(uint8_t)*((nTriangles + 7)/8));
125     memset(triangleScore, 0, sizeof(ScoreType)*nTriangles);
126     memset(triangleIndices, 0, sizeof(TriangleIndexType)*3*
            nTriangles);
127
128     // Count the triangle array offset for each vertex,
129     // initialize the rest of the data.
130     int sum = 0;
131     for (int i = 0; i < nVertices; i++) {
132         offsets[i] = sum;
133         sum += numActiveTris[i];
134         numActiveTris[i] = 0;
135         cacheTag[i] = -1;
136     }
137
138     // Fill the vertex data structures with indices to the triangles
139     // using each vertex
140     for (int i = 0; i < nTriangles; i++) {
141         for (int j = 0; j < 3; j++) {
142             int v = indices[3*i + j];
143             triangleIndices[offsets[v] + numActiveTris[v]] = i;
144             numActiveTris[v]++;
145         }
146     }
147
148     // Initialize the score for all vertices
149     for (int i = 0; i < nVertices; i++) {
150         lastScore[i] = findVertexScore(numActiveTris[i], cacheTag[i
                ]);
151         for (int j = 0; j < numActiveTris[i]; j++)
152             triangleScore[triangleIndices[offsets[i] + j]] +=
                    lastScore[i];
153     }
154
155     // Find the best triangle
156     int bestTriangle = -1;
157     int bestScore = -1;
158
```

```
159     for (int i = 0; i < nTriangles; i++) {
160         if (triangleScore[i] > bestScore) {
161             bestScore = triangleScore[i];
162             bestTriangle = i;
163         }
164     }
165
166     // Allocate the output array
167     TriangleIndexType* outTriangles = new TriangleIndexType[
            nTriangles];
168     int outPos = 0;
169
170     // Initialize the cache
171     int cache[VERTEX_CACHE_SIZE + 3];
172     for (int i = 0; i < VERTEX_CACHE_SIZE + 3; i++)
173         cache[i] = -1;
174
175     int scanPos = 0;
176
177     // Output the currently best triangle, as long as there
178     // are triangles left to output
179     while (bestTriangle >= 0) {
180         // Mark the triangle as added
181         SETADDED(bestTriangle);
182         // Output this triangle
183         outTriangles[outPos++] = bestTriangle;
184         for (int i = 0; i < 3; i++) {
185             // Update this vertex
186             int v = indices[3*bestTriangle + i];
187
188             // Check the current cache position, if it
189             // is in the cache
190             int endpos = cacheTag[v];
191             if (endpos < 0)
192                 endpos = VERTEX_CACHE_SIZE + i;
193             // Move all cache entries from the previous position
194             // in the cache to the new target position (i) one
195             // step backwards
196             for (int j = endpos; j > i; j--) {
197                 cache[j] = cache[j-1];
198                 // If this cache slot contains a real
199                 // vertex, update its cache tag
200                 if (cache[j] >= 0)
201                     cacheTag[cache[j]]++;
202             }
```

```
203             // Insert the current vertex into its new target
204             // slot
205             cache[i] = v;
206             cacheTag[v] = i;
207 
208             // Find the current triangle in the list of active
209             // triangles and remove it (moving the last
210             // triangle in the list to the slot of this triangle).
211             for (int j = 0; j < numActiveTris[v]; j++) {
212                 if (triangleIndices[offsets[v] + j] == bestTriangle)
                        {
213                     triangleIndices[offsets[v] + j] =
214                         triangleIndices[
215                             offsets[v] + numActiveTris[v] - 1];
216                     break;
217                 }
218             }
219             // Shorten the list
220             numActiveTris[v]--;
221         }
222         // Update the scores of all triangles in the cache
223         for (int i = 0; i < VERTEX_CACHE_SIZE + 3; i++) {
224             int v = cache[i];
225             if (v < 0)
226                 break;
227             // This vertex has been pushed outside of the
228             // actual cache
229             if (i >= VERTEX_CACHE_SIZE) {
230                 cacheTag[v] = -1;
231                 cache[i] = -1;
232             }
233             ScoreType newScore = findVertexScore(numActiveTris[v],
234                                                  cacheTag[v]);
235             ScoreType diff = newScore - lastScore[v];
236             for (int j = 0; j < numActiveTris[v]; j++)
237                 triangleScore[triangleIndices[offsets[v] + j]] +=
                        diff;
238             lastScore[v] = newScore;
239         }
240         // Find the best triangle referenced by vertices in the
            cache
241         bestTriangle = -1;
242         bestScore = -1;
243         for (int i = 0; i < VERTEX_CACHE_SIZE; i++) {
244             if (cache[i] < 0)
```

```
245                 break;
246             int v = cache[i];
247             for (int j = 0; j < numActiveTris[v]; j++) {
248                 int t = triangleIndices[offsets[v] + j];
249                 if (triangleScore[t] > bestScore) {
250                     bestTriangle = t;
251                     bestScore = triangleScore[t];
252                 }
253             }
254         }
255         // If no active triangle was found at all, continue
256         // scanning the whole list of triangles
257         if (bestTriangle < 0) {
258             for (; scanPos < nTriangles; scanPos++) {
259                 if (!ISADDED(scanPos)) {
260                     bestTriangle = scanPos;
261                     break;
262                 }
263             }
264         }
265     }
266
267     // Clean up
268     delete [] triangleIndices;
269     delete [] offsets;
270     delete [] lastScore;
271     delete [] numActiveTris;
272     delete [] cacheTag;
273     delete [] triangleAdded;
274     delete [] triangleScore;
275
276     // Convert the triangle index array into a full triangle list
277     VertexIndexType* outIndices = new VertexIndexType[3*nTriangles];
278     outPos = 0;
279     for (int i = 0; i < nTriangles; i++) {
280         int t = outTriangles[i];
281         for (int j = 0; j < 3; j++) {
282             int v = indices[3*t + j];
283             outIndices[outPos++] = v;
284         }
285     }
286     delete [] outTriangles;
287
288     return outIndices;
289 }
```