LPT Codes used with ROAM Splitting Algorithm

CMSC451 Honors Project

Joseph Brosnihan

Mentor: Dave Mount

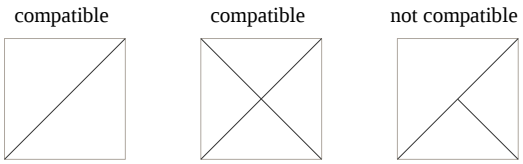
Fall 2015

Abstract

Simplicial meshes are commonly used to represent regular triangle grids. Of many areas, simplicial meshes are useful in the area of terrain rendering within computer graphics. In this project, I write an implementation of Atalay and Mount's LPT code representation of simplices. I implement part of the ROAM terrain algorithm that splits simplices down to a desired level of detail. I make several modifications to the LPT code representation. I test these varying representations in several scenarios.

Introduction

The ROAM algorithm uses simplex decompositions to render a mesh in real time with an adapting level of detail. The algorithm uses a priority queue of all currently visible triangles, sorted in order of error (where error is some numerical metric for visual fidelity). The algorithm repeatedly splits the triangle with the highest error to increase visual fidelity, while merging triangles with the lowest error to increase performance. ROAM maintains that the visible triangulation is always compatible, that is, no triangle has an edge bisected by a vertex of another triangle.



My original goal of this project was to implement and optimize the triangle splitting and merging processes of ROAM. When I learned about pointerless triangle representations, my goal shifted to implementing and optimizing these pointerless representations.

LPT Codes

Atalay and Mount introduce the concept of an LPT code, an encoding that defines a unique simplex in a simplex decomposition, in their paper *Pointerless Implementation of Hierarchical Simplicial Meshes and Efficient Neighbor Finding in Arbitrary Dimensions*. As they define it, an LPT code consists of three things: the level, permutation, and translation. The level is a number that defines the depth of the simplex in the simplex decomposition tree. The permutation is a linear transformation that when applied to the base simplex will give the orientation of the specific simplex represented by the LPT code. The permutation is represented by a sequence $\Pi=\{p_1, ..., p_d\}$. The i^{th} column of the matrix of the corresponding linear transformation is given by $\text{sign}(\Pi[i]) \cdot e_{|\Pi[i]|}$ where e_j is the vector whose j^{th} entry is 1 and other entries are 0. The translation, known as the orthant list, is a list of vectors that describe the path down the simplex decomposition tree to the specific simplex. The sum of each orthant in the orthant list, where the i^{th} orthant in the list is multiplied by $(½)^i$, will give the location of the specific simplex.

When working with a simplex decomposition tree, one often needs to be able to traverse the tree by finding the parent, child, or neighbor of a given simplex. This is the case with the ROAM algorithm. Atalay and Mount present these operations on LPT codes in their paper. These operations are not conceptually complicated, but they involve handling many cases. These operations are covered in detail in their paper.

LPT codes provide a pointerless representation of simplices. These operations that traverse a simplex decomposition tree can be performed using only the input LPT code without accessing an external data structure (i.e. without pointers). The output LPT code may be used as an index into a hash table, for example, to retrieve information associated with the corresponding simplex. In my case, I used a hash table to look up whether a given simplex was part of the current triangulation.

Implementation

All of my code is available on Github at github.com/JoeBrosnihan/ROAM-Terrain

I wrote my implementation in C++. It builds on Unix. To represent triangles, I use a `struct`. I use two alternate `structs` to represent a triangle LPT code: one in which the parameters are stored as `ints`, and another in which the parameters are packed tightly into a binary string. I explain the differences more below. The LPT codes for all active triangles are each stored in a vector for quick iteration, and in a hash table (C++ unordered_set) for quick checks to see if a triangle is active. I implemented ROAM's "force split" algorithm which splits a triangle, maintaining the compatibility of the triangulation by recursively splitting neighbors if needed.

I set up several tests to compare the performance of when the LPT code parameters are stored packed into a bit string versus unpacked as `ints`. My hypothesis was that packing LPT code parameters would save space, but require more CPU clock cycles to unpack and process compared to unpacked parameters. Under my hypothesis, a smaller `struct` would allow more LPT codes to fit in the CPU cache, leading to an amortized increase in performance from fewer accesses to slower levels of memory, such as RAM and lower levels of cache.

Here are the packed and non-packed versions of the lptcode struct, respectively:

```
struct lptcode {
        //The length of the simplex code
        int len_p;
        //l = |p| mod d, the simplex's level
        int l;
        //Permutation stored as {+/-1, +/-2}
                // or {+/-2, +/-1}
        int permutation[2];
        //list of orthants (always +/-1, +/-1)
        int orthant_list[MAX_ORTH_LIST_LEN * 2];
}
```

```
struct lptcode {
        //the length of the simplex code
        uint8_t len_p;
        //permutation is 1,2 if bit0 is 0; 2,1 if
                //bit0 is 1
        //first entry of transform is positive iff
                //bit1 is 0
        //second entry ... bit2
        //starting with bit4, every two bits store
        //the sign of the orthant's x and y coords
        //respectively. 0 => positive, 1 => neg
        uint8_t data[DATA_LEN];
}
```

- `len_p` is the length of the binary string that represents the simplex. It's length is equal to the number of times the simplex has been split (i.e. its depth in the decomposition tree).
- `MAX_ORTH_LIST_LEN` is a predefined integer that determines how many orthants an LPT code struct must be able to hold in its orthant list. This determines the size of the struct.
- The `DATA_LEN` seen above is a predefined integer for the packed struct only equal to the number of bytes the struct must store for the permutation and orthant list. `DATA_LEN` is defined to be the ceiling of `(4 + 2 * MAX_ORTH_LIST_LEN) / 8`.

The unpacked struct takes up `16+8*MAX_ORTH_LIST_LEN` bytes. The packed struct takes up `4+2*MAX_ORTH_LIST_LEN` ***bits*** rounded up to the nearest byte.
Note that for even for the small value of `MAX_ORTH_LIST_LEN=10`, the packed struct takes 4 bytes while the non-packed struct takes 96 bytes.

Experiment

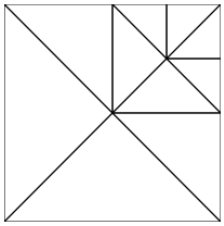
I constructed a test scenario to split a triangle mesh in the same way that ROAM's split process would. Every time a triangle is split, its children are checked against a function `needs_split(struct lptcode)`, which returns true if the child needs to be split further to reach the test's target level of detail. Triangles that need to be split are put in a queue. The program repeatedly splits the triangle at the head of the queue until the queue is empty.

There are several parameters for each test:

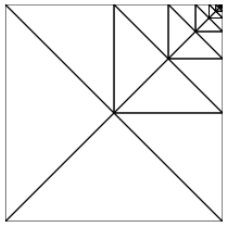
- Whether the LPT codes are Non-Packed or Packed
- The criteria to split a triangle (i.e. the `needs_split` function)
- The value of `MAX_ORTH_LIST_LEN`

I created three different `needs_split` functions, each of which splits some subset of triangles down to a given level of detail. This is a predefined integer, `TARGET_LOD`, which I varied in the tests. Here are the three types of `needs_split` functions I used and some examples:

Test Corner – Every triangle touching the upper right corner of the triangulation is split, until the only triangles touching it have been split `TARGET_LOD` number of times.

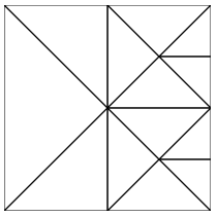


TARGET_LOD = 4

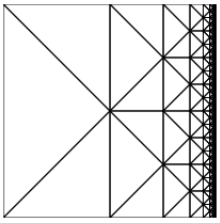


TARGET_LOD = 32

Test Wall – Every triangle touching the rightmost edge of the triangulation is split, until the only triangles touching it have been split `TARGET_LOD` number of times.

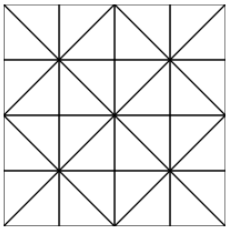


TARGET_LOD = 4

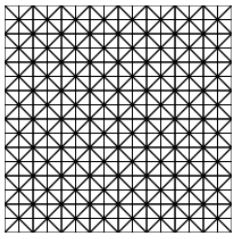


TARGET_LOD = 16

Test Constant – Every triangle is split until all triangles have been split `TARGET_LOD` number of times.



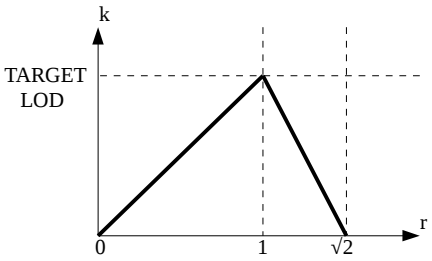
TARGET_LOD = 4

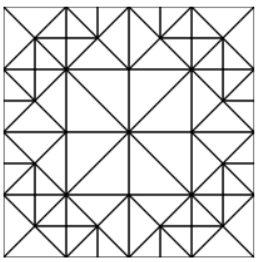


TARGET_LOD = 8

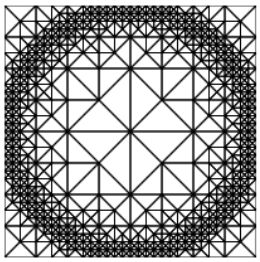
Test Circle – Triangles are split to a level of detail based on how close they are to the border of a unit circle.

Triangles are checked against a function that takes in the x, y coordinates of a triangle's centroid and outputs an integer k. If the triangle has been split less than k times, it needs to be split further. In this test, k is a function of r, the distance from (x, y) to the center of the square, where the square is 2 units long along one side. The function k(r) is shown.





TARGET_LOD = 7



TARGET_LOD = 12

Test Corner Results

`MAX_ORTH_LIST_LEN` set to a constant value of 32 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
32	0.000663665	0.000405657
48	0.00104406	0.000403242
56	0.00127275	0.00064073
60	0.00147073	0.000815593
64	0.0015829	0.00093135

`MAX_ORTH_LIST_LEN` set to a constant value of 126 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
32	0.000926171	0.000609156
64	0.00200532	0.00105725
128	0.00584485	0.00319287
192	0.00833885	0.00569061
252	0.0114909	0.00758006

Test Wall Results

`MAX_ORTH_LIST_LEN` set to a constant value of 16 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
12	0.00230837	0.00165184
16	0.0162633	0.00911186
20	0.133308	0.0839384
24	2.08587	1.11794
26	11.6616	4.36122
28	53.1481	17.227

`MAX_ORTH_LIST_LEN` set to a constant value of 32 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
12	0.00242425	0.00152279
16	0.0195514	0.0111852
20	0.175738	0.0856738
24	3.94088	1.1535
26	19.4543	4.50609
28	84.3511	17.8824

Test Constant Results

`MAX_ORTH_LIST_LEN` set to a constant value of 8 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
8	0.00404213	0.00210322
10	0.0276484	0.0302133
12	0.289768	0.0401933
14	6.30998	0.301363
15	36.9348	0.980773
16	168.135	3.44733
17	N/A	12.6146
18	N/A	13.1483

`MAX_ORTH_LIST_LEN` set to a constant value of 32 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
8	0.00693997	0.00255633
10	0.0672615	0.0104625
12	1.07764	0.0667466
14	34.9159	0.574979
15	146.21	2.04815
16	593.212	7.70545
17	N/A	45.1937
18	N/A	290.965

Test Circle Results

`MAX_ORTH_LIST_LEN` set to a constant value of 16 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
12	0.0206431	0.0101204
14	0.182777	0.0667745
16	1.95789	0.635462
17	11.924	2.13183
18	47.8254	7.43074
19	186.189	26.4802
20	N/A	94.7061

`MAX_ORTH_LIST_LEN` set to a constant value of 32 for these tests.

TARGET_LOD	Non-packed time (sec)	Packed time (sec)
12	0.0335484	0.013921
14	0.271488	0.0713894
16	5.49222	0.670409
17	23.4691	2.27119
18	84.1936	7.81761
19	322.863	28.15
20	N/A	105.603

Conclusions

Overall, the packed LPT code structs outperformed the non-packed structs by far in every test case. This is primarily due to the size of the packed structs. The benefit of small size is apparent when we examine the effects of padding the structs with extra space for unused orthants.

In the Wall test with TARGET_LOD = 28, Non-packed structs took 58% longer when MAX_ORTH_LIST_LEN was changed from 16 to 32. Packed structs took only 3.8% longer.
In the Constant test with TARGET_LOD = 16, Non-packed structs took 252% longer when MAX_ORTH_LIST_LEN was changed from 8 to 32. Packed structs took only 123% longer.

There are several memory related reasons for this speedup. 1) smaller structs means more structs can fit in fast levels of memory at once, requiring fewer accesses to slower levels of memory on average over time. 2) LPT codes are well suited to be compact. For example, if the compiler is smart enough to make this optimization, the bytes containing an LPT code's permutation and orthant list can be loaded into a single register in the CPU, in which most or all of the orthant list can be processed, changed, perhaps even compared to another LPT orthant list, without making a single memory access. This is usually not possible if the orthant list is stored as an array.

I realized when writing the packed LPT code implementation that there weren't as many extra instructions as I thought. In fact, I found that many if-statements in my non-packed implementation could be replaced with bit manipulations that were not conditional. This, perhaps, also had some benefit on performance through avoiding conditional jumps.

References

1. F. B. Atalay and D. M. Mount, Pointerless implementation of hierarchical simplicial meshes and efficient neighbor finding in arbitrary dimensions, Proc. 13th Int. Meshing Roundable, 2004.
2. M. Duchaineau et al, ROAMing terrain: real-time optimally adapting meshes, Proc. 8th conference on Visualization, 1997.
3. J. M. Maubach, Local bisection refinement for N-simplicial grids generated by reflection, SIAM J. Sci. Stat. Comp. 16 (1995) 210-227.
4. J. M. Maubach, The efficient location of neighbors for locally refined n-simplicial grids, Proc. 5th Int. Meshing Roundable, 1996.
5. T. Chilimbi, M. D. Hill and J. R. Larus, Cache-conscious structure layout, in Programming Languages Design and Implementation, 1999.