3D Software Rasterisation Renderer
Com S 424 Final Project

Ian Malerich

imm@iastate.edu

December 4, 2017

Introduction

This project implements a rendering system for 3 dimensional graphics, defined as a set of triangles in 3 dimensional space. Models are input via a .obj file containing triangle data as well as a .mtl referenced by the .obj file which holds texture information. A camera is set at the origin looking down the z-axis and captures a single image representing the scene. Therefore the problem size varies across two primary variables: the number of triangles which need to be rendered and the dimensions of the output texture, that is, how many pixel need to be colored.

Systems

Results are included for both my personal system and the hpc cluster. The below chart provides some basic information about each system as read from /proc/cpuinfo.

system	personal	hpc-class
model name	i7-3630QM	Xeon CPU-E5-2650
cores	4	8
processors	8	16
clock speed	2.40GHz	2.00GHz
cache size	6144 KB	20480 KB
address size (physical)	36 bits	46 bits
address size (virtual)	48 bits	48 bits

Models

Below are a list of models which will be used for various performance tests as well as how many vertices make up the model. Note that each model has accompanying textures which will affect memory usage but in terms of performance texture samples are handled per pixel and thus the output image size is the bigger concern and not buffer size. It is worth mentioning that while the model 'war' has the least vertices, it is the only model not made up of a single mesh. Rather, war is made up by 10 smaller models each with their own texture. Thus as the provided graphs will demonstrate the performance of this mesh does not quite fall in line with the other models.

war   dragon

blackdragon   ironman
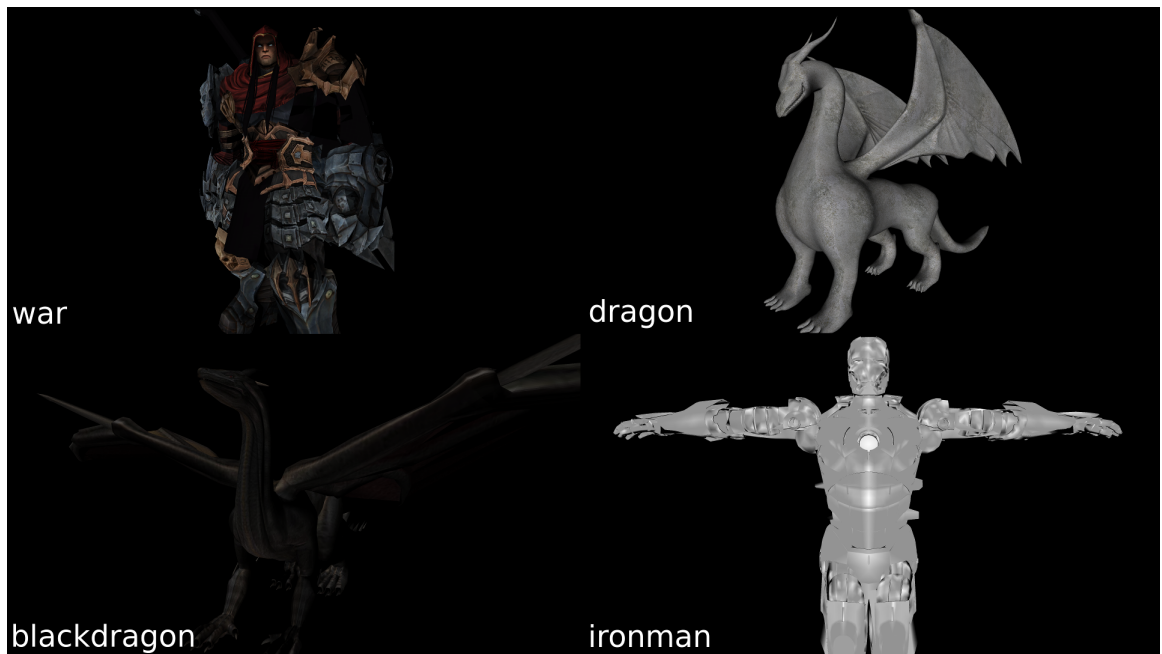
name	vertex count	parts
war	44,115	10
dragon	61,152	1
blackdragon	113,955	1
iron man	596,682	1

Measurements

The primary goal in regards to performance is to minimize total rendering time. To that respect, I will not be including model and texture load times as well as output image write time in any measurements. The clock will begin at the start at the rendering process and end once rendering is complete.

Serial Execution

Below we can see how well my code performs when running in serial. We can clearly see that the output image resolution has a much greater influence on total render time. Number of vertices does tend to produce longer render times but the increase is much more gradual. Note that an output image of size 3840×2160 (orange) is exactly $4\times$ that of 1920×1080 (blue) thus based only on results so far render time appears to scale almost linearly with image resolution. Regardless, based on this graph our concern is much less with vertex transformations and much more with pixel rendering.
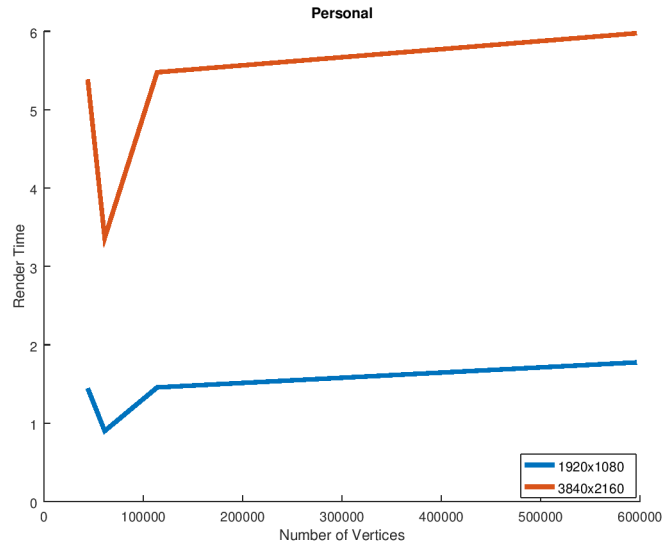
I ran the valgrind profiler 'callgrind' on the blackdragon mesh at a resolution of 3840×2160 to see exactly how much time was spent in code. The table below contains relevant data produced by callgrind. Note I have ommitted calls involved in model/texture loading which require compressions/disk access beyond my control and concern.

Ir	Function	Primary Use
2,927,228,239	isPointInTriangle	Should a pixel be drawn for a give triangle.
2,793,061,201	vecCross	Find bary centric coords for vertex interpolation.
1,739,333,716	drawTriangle	Interplate data and fill pixels.
1,239,622,129	sampleNearest	Sample a color from a texture.
647,909,164	sampleLinear	Interpolate 4 nearest samples from a texture.
509,067,930	roundf	Round coordinates for texture sampling.
426,630,011	simpleShader()	Color in a given pixel.

All top calls which influence the measurements I am taking directly relate
pixel rendering and not vertex transformations. Confirming what we noted
about the serial execution graph. This tells us that while improved vertex
transformations may help our performance, the bigger gains are to be made
from parallelizing the pixel rendering code.

Parallelization

The easiest method, and the first method I will try, to parallelize this code
is a simple parallel for directive loop over each face. Each thread will then
transform 3 vertices corresponding to its assigned face, and then draw the
given triangle. Within the draw_triangle method critical sections are needed
when writing to the back and output buffers so that only one thread may
write at a time. All other operations may be performed in parallel.

```
1   #pragma omp parallel for num_threads(thread_count)
2   for (int i=0; i<num_faces(m); i++) {
3       transform_vertex(&m.vertices[3*i + 0], &proj, WIDTH, HEIGHT);
4       transform_vertex(&m.vertices[3*i + 1], &proj, WIDTH, HEIGHT);
5       transform_vertex(&m.vertices[3*i + 2], &proj, WIDTH, HEIGHT);
6
7       draw_triangle(&m, i, simple_shader, m.material, buffer,
```

```
8      back_buffer, SIZE);
9  }
```

```
1  #pragma omp critical
2  {
3    if (back[y*buffer_size.x + x] > pos.z) {
4      back[y*buffer_size.x + x] = pos.z;
5
6      // interpolate vertex data
7      vector_t tex_coord = interpolate(tex_coords,
8        bc_screen);
9      vector_t norm = interpolate(norms, bc_screen);
10     vector_t tan = interpolate(tans, bc_screen);
11
12     vector_t c = shader((shader_data_t){pos,
13       tex_coord, norm, tan}, material);
14     draw_point(vector_to_point(pos),
15       vector_to_color(c), buffer, buffer_size);
16   }
17 }
```

The immediate issue with this is the giant critical section surrounding the
pixel rendering code. We found by analyzing the serial implementation that
this is where the majority of our work will take place and by forcing it into
a critical section we are severely limiting the potential performance gains.
Nevertheless, we still see a performance increase when introducing
threading, although limited to just a few threads. Additional threads likely
do not provide enough added benefit to overcome the limitations of the per
pixel critical section.

Personal
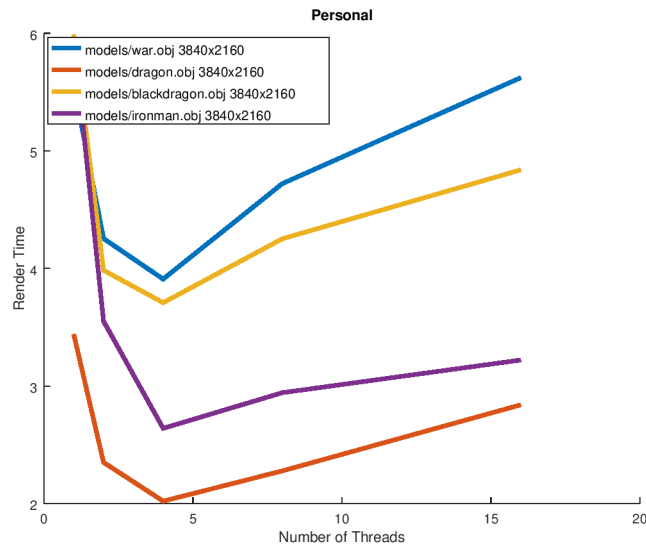
Note that vertex transformation code is entirely independent each triangle. As such, we can split our vertex transformation code and triangle rasterization code into separate loops. This exemplifies the idea that these are the two dimensions on which the program scales. By doing this if a thread would be stuck waiting for a critical section in the old code, it can now continue to transform vertices in parallel. By using this method, we can see a very slight but insignificant increase in performance, particularly when number of threads is large.
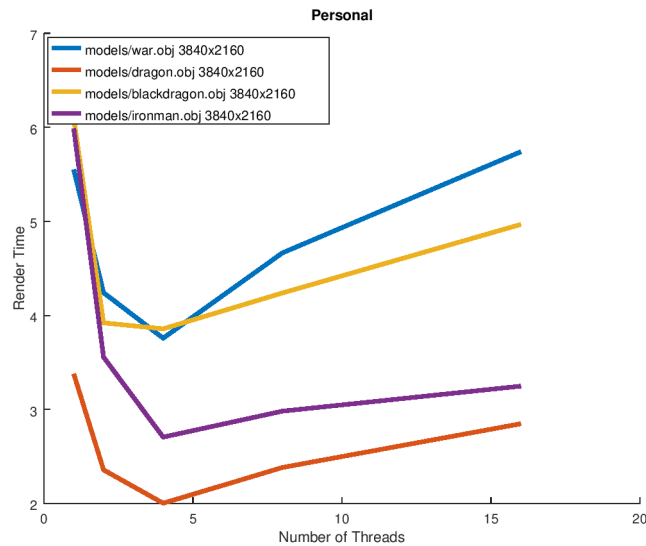
```
1   #pragma omp parallel for num_threads(thread_count)
2   for (int i=0; i<m.vert_count; i++) {
3     transform_vertex(&m.vertices[i], &proj, WIDTH, HEIGHT);
4   }
5
6   #pragma omp parallel for num_threads(thread_count)
7   for (int i=0; i<num_faces(m); i++) {
```

```
8    draw_triangle(&m, i, simple_shader, m.material, buffer,
9      back_buffer, SIZE);
10  }
```


To get over the critical section hurdle, we could introduce a lock for each given pixel. We can have multiple threads writing to buffers so long as they are not writing to the same location (triangles can overlap!). However locks can be slow and for a 3840×2160 image we would require 8.3 million locks, which would need a lock and unlock for every single write. This seems highly impractical and not likely to solve anything. Thus I will ignore this strategy.

A much better approach would be to run the rasterization step in parallel. That is, we loop through triangles on our main thread, and then spawn a set of threads which iterate over the bounding region of that triangle. Thus each per pixel operation can be performed in parallel. Further since each pixel within a bounding box is unique, we can guarantee each thread is

reading/writing to a unique location in the back and output buffers. Thus no critical section is necessary in the below implementation.

```
1  #pragma omp parallel for num_threads(thread_count)
2  for (int i=0; i<m.vert_count; i++) {
3    transform_vertex(&m.vertices[i], &proj, WIDTH, HEIGHT);
4  }
5
6  for (int i=0; i<num_faces(m); i++) {
7    draw_triangle(&m, i, simple_shader, m.material, buffer,
8      back_buffer, SIZE);
9  }
```
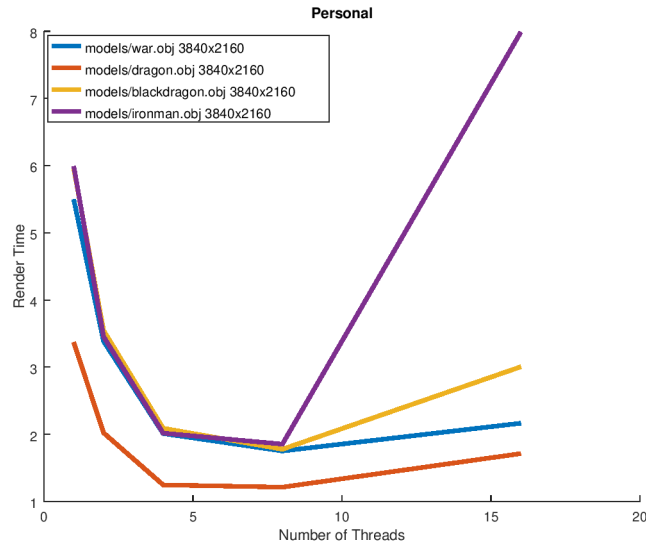
```
1  #pragma omp parallel for num_threads(thread_count)
2  for (int idx = 0; idx < count; idx++) {
3    int x = (idx % width) + bbox.min.x;
4    int y = (idx / width) + bbox.min.y;
5
6    vector_t pos = make_vector(x, y, 0, 0);
7
8    if (is_point_in_triangle(pos, vertices)) {
9      // compute bary centric coordinates,
10     // we will need this for our depth test
11     vector_t bc_screen =
12       bary_centric(make_vector(x,y, 0.0, 0.0), vertices);
13     if (bc_screen.x < 0 || bc_screen.y < 0 || bc_screen.z < 0)
14       { continue; }
15     pos.z = (vertices[0].z * bc_screen.x) +
16       (vertices[1].z * bc_screen.y) +
17       (vertices[2].z * bc_screen.z);
18
```

```
19    if (back[y*buffer_size.x + x] > pos.z) {
20      back[y*buffer_size.x + x] = pos.z;
21
22      // interpolate vertex data
23      vector_t tex_coord =
24        interpolate(tex_coords, bc_screen);
25      vector_t norm = interpolate(norms, bc_screen);
26      vector_t tan = interpolate(tans, bc_screen);
27
28      vector_t c = shader((shader_data_t){pos, tex_coord,
29        norm, tan}, material);
30      draw_point(vector_to_point(pos), vector_to_color(c),
31        buffer, buffer_size);
32    }
33  }
34 }
```

In the graph below we can see much larger increases in performance than we
have experienced before. Even the most complex models can be rendered in
just over 2 seconds with the simplest approaching 1 second. Note that on
my personal system a huge spike upwards occurs for > 8 threads, this is not
particularly of interest as my system cannot run any more than 8 threads
at a given time thus using more only incurs additional overhead.

Personal

Legend:
- models/war.obj 3840x2160
- models/dragon.obj 3840x2160
- models/blackdragon.obj 3840x2160
- models/ironman.obj 3840x2160

X-axis: Number of Threads
Y-axis: Render Time

Method Comparison

In all cases, ironman took the most significant amount of time to render. The graph below shows a comparison between the 3 methods in rendering this model at 3840×2160. We can see the final solution branching off from the other methods starting particularly at 4+ threads. Interestingly, we can see Method 3 is hit the hardest when run on too many threads, though clearly in all other cases it performs the best.

Personal
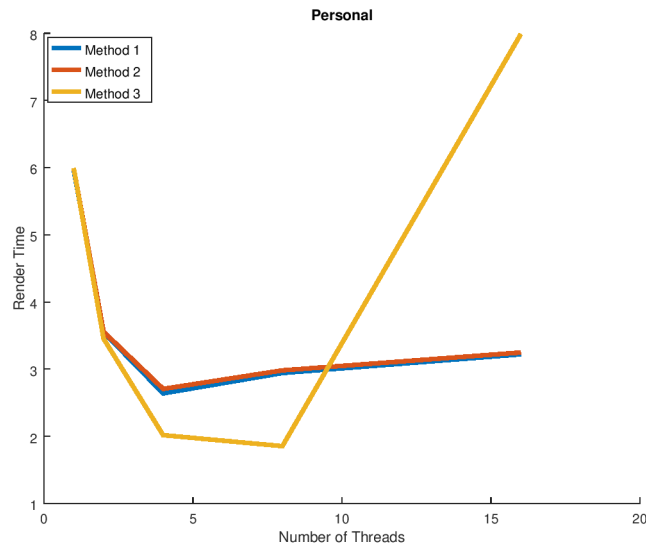
OpenMP Scheduling

Using Method 3, the following graphs analyze different OpenMP scheduling methods and chunk sizes. With the exception of (dynamic, 1) performing notably terrible, they all seem relatively close in performance to each other and all follow the same overall trends. It appears larger chunk sizes provide the best performance. This is likely due to the fact that each operation is relatively short and thus threads will benefit having nearby memory for their next computation, a trait that a larger chunk size provides.
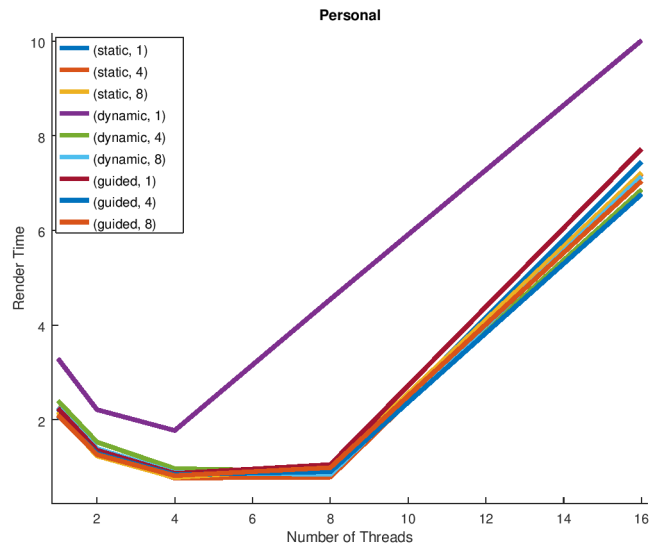
Running static distrubition with various chunk sizes shows that the chunk size can clearly be too large. Sizes 64 and 128 are particularly terrible. Chunk sizes between 4 and 32 seem to perform the best.
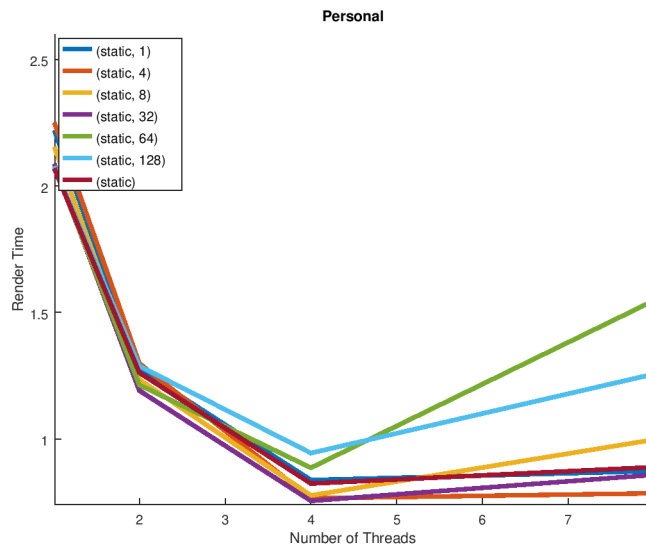
Conclusion

References

- Lengyel, Eric. *Mathematics for 3D Game Programming and Computer Graphics.* Course Technology, Cengage Learning, 2012.

- Sokolov, Dmitry. "Tinyrenderer". https://github.com/ssloy/tinyrenderer/wiki

- Barrett, Sean. "stb". https://github.com/nothings/stb

- Michael SüB and Claudia Leopold. *Common Mistakes in OpenMP and How To Avoid Them.*