3D Software Rasterisation Renderer

Com S 424 Final Project

Ian Malerich

imm@iastate.edu

December 5, 2017

Introduction

This project implements a rendering system for 3D computer 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. The image below was output by this program given the input file 'models/war.obj'.

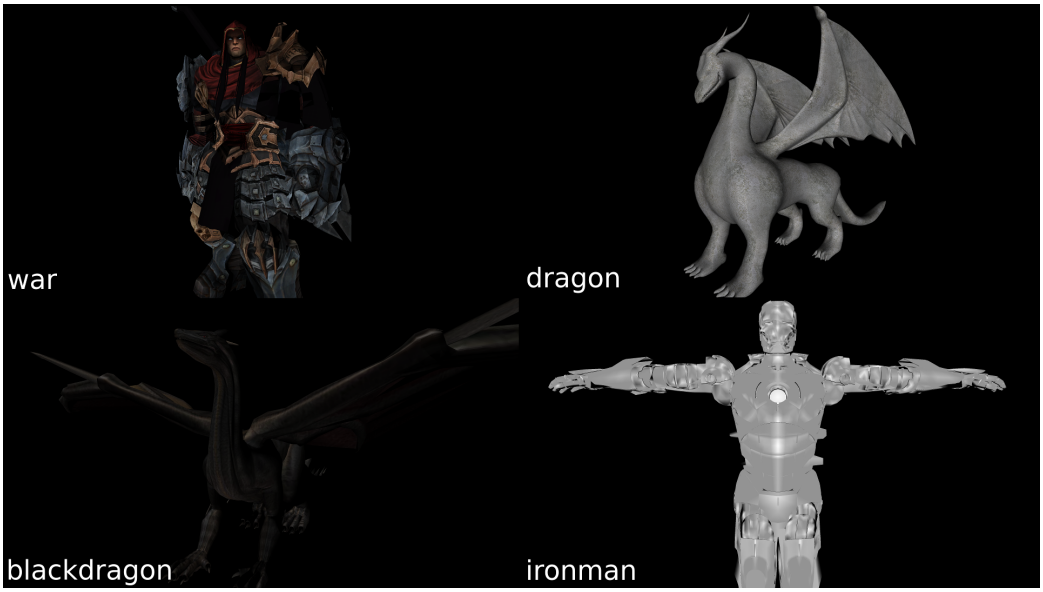
Systems

Results are included for both my personal system and the hpc cluster. The chart below 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. However, with respect to performance, texture samples are handled per pixel, and thus the size of the output image is the bigger concern and not texture 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 independent set of vertices and texture. Thus as the provided graphs will demonstrate the performance of this mesh does not quite fall in line with the other models with respect to vertex count.



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

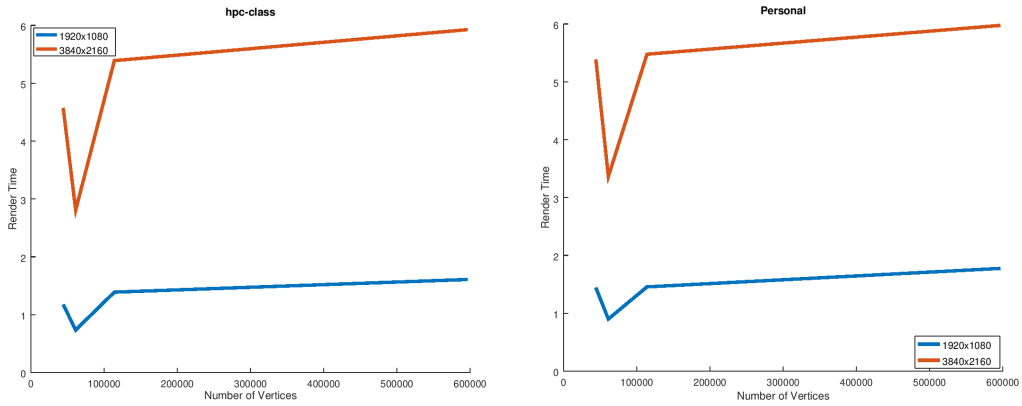
Measurements

The primary performance goal 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. Thus, factors such as disk speed will not have an impact on my results.

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 to 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

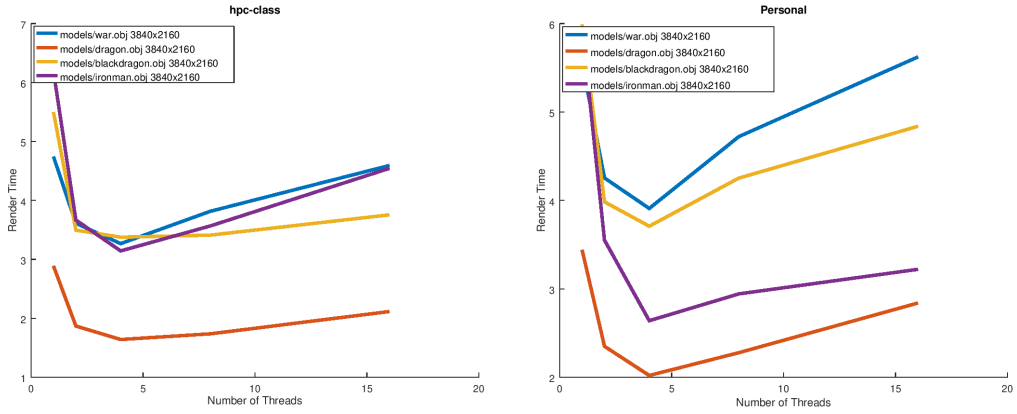
The easiest method, and the first method I will try, to parallelize this code is a simple parallel for loop directive over each face. Each thread will then transform 3 vertices corresponding to its assigned face, and then draw the given triangle. Within the draw_triangle method critical sections are needed when writing to the back and output buffers so that only one thread may write at a time. This is necessary as it is possible for triangles to overlap on screen. 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
```

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

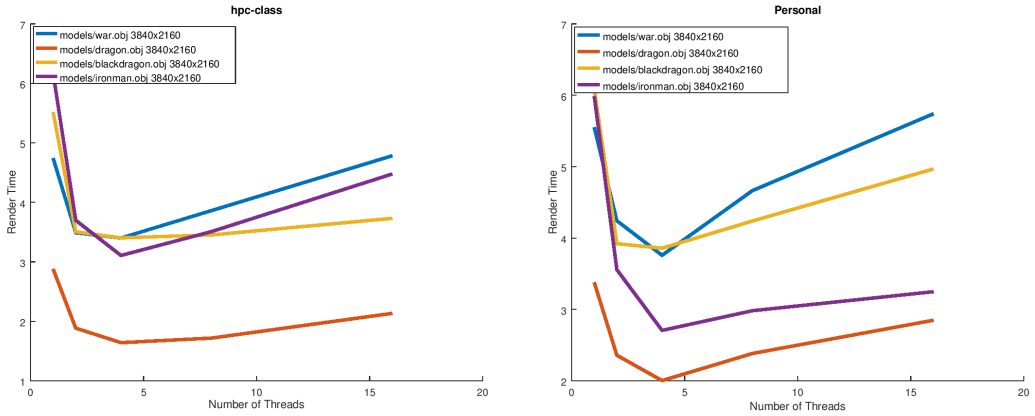
```
#pragma omp critical
{
  if (back[y*buffer_size.x + x] > pos.z) {
    back[y*buffer_size.x + x] = pos.z;

    // interpolate vertex data
    vector_t tex_coord = interpolate(tex_coords,
      bc_screen);
    vector_t norm = interpolate(norms, bc_screen);
    vector_t tan = interpolate(tans, bc_screen);

    vector_t c = shader((shader_data_t){pos,
      tex_coord, norm, tan}, material);
    draw_point(vector_to_point(pos),
      vector_to_color(c), buffer, buffer_size);
  }
}
```

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. Note that both my personal system and the hpc-class system both reach minimum render time using 4 threads. Further threads decrease performance.



Each vertex transformation is entirely independent from triangle rasterisation. As such, we can split our vertex transformation code and triangle rasterisation 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. Because the added benefit is within vertex transformations, and not pixel rendering, almost no difference is seen between this method and our previous approach.

```
#pragma omp parallel for num_threads(thread_count)
for (int i=0; i<m.vert_count; i++) {
  transform_vertex(&m.vertices[i], &proj, WIDTH, HEIGHT);
}

#pragma omp parallel for num_threads(thread_count)
for (int i=0; i<num_faces(m); i++) {
  draw_triangle(&m, i, simple_shader, m.material, buffer,
    back_buffer, SIZE);
}
```



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 (recall triangles can overlap). However locks can be slow and for a 3840×2160 image we would require 8.3 million locks, which would require a lock and unlock operation for every single read/write. This seems highly impractical and not likely to solve anything.

A much better approach would be to run the rasterisation 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. Therefore no critical section is necessary for this implementation.

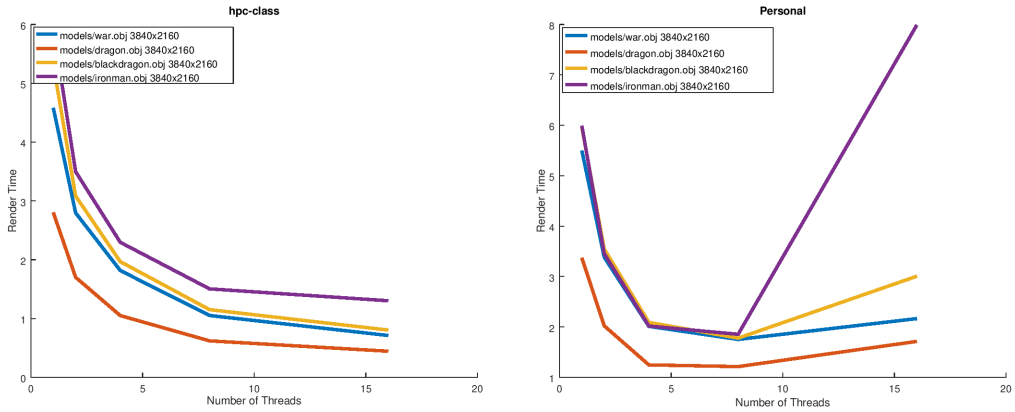
```
1  #pragma omp parallel for num_threads(thread_count)
2  for (int i=0; i<m.vert_count; i++) {
3    transform_vertex(&m.vertices[i], &proj, WIDTH, HEIGHT);
4  }
5
```

```
for (int i=0; i<num_faces(m); i++) {
  draw_triangle(&m, i, simple_shader, m.material, buffer,
    back_buffer, SIZE);
}
```

```
#pragma omp parallel for num_threads(thread_count)
for (int idx = 0; idx < count; idx++) {
  int x = (idx % width) + bbox.min.x;
  int y = (idx / width) + bbox.min.y;

  vector_t pos = make_vector(x, y, 0, 0);

  if (is_point_in_triangle(pos, vertices)) {
    // compute bary centric coordinates,
    // we will need this for our depth test
    vector_t bc_screen =
      bary_centric(make_vector(x,y, 0.0, 0.0), vertices);
    if (bc_screen.x < 0 || bc_screen.y < 0 || bc_screen.z < 0)
      { continue; }
    pos.z = (vertices[0].z * bc_screen.x) +
      (vertices[1].z * bc_screen.y) +
      (vertices[2].z * bc_screen.z);

    if (back[y*buffer_size.x + x] > pos.z) {
      back[y*buffer_size.x + x] = pos.z;

      // interpolate vertex data
      vector_t tex_coord =
        interpolate(tex_coords, bc_screen);
      vector_t norm = interpolate(norms, bc_screen);
      vector_t tan = interpolate(tans, bc_screen);
```

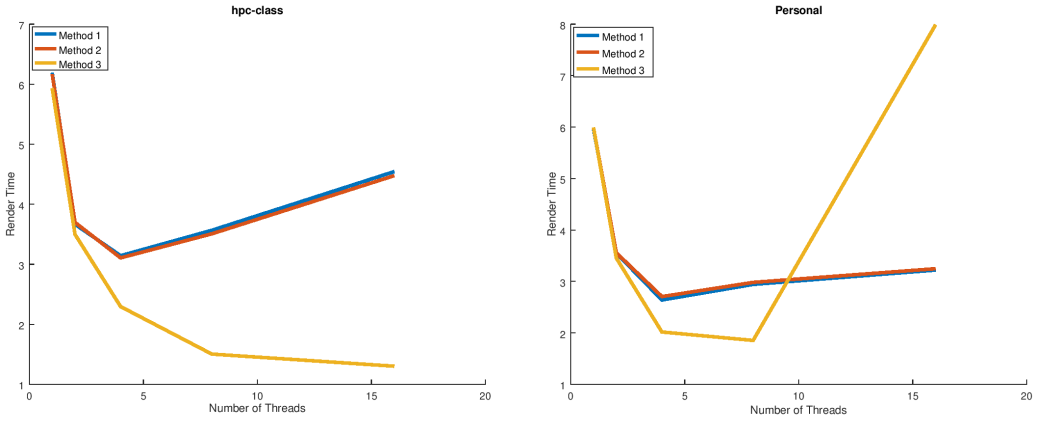
```
27
28          vector_t c = shader((shader_data_t){pos, tex_coord,
29            norm, tan}, material);
30          draw_point(vector_to_point(pos), vector_to_color(c),
31            buffer, buffer_size);
32        }
33      }
34  }
```

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 near 1 second or less. Note that on my personal system a major 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. The hpc-class system, for which 16 threads are available, continued to see a small performance increase from 8 to 16 threads.



Method Comparison

The graph below shows a comparison between the 3 methods in rendering 'models/ironman.obj' at a resolution of 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.



Memory Usage

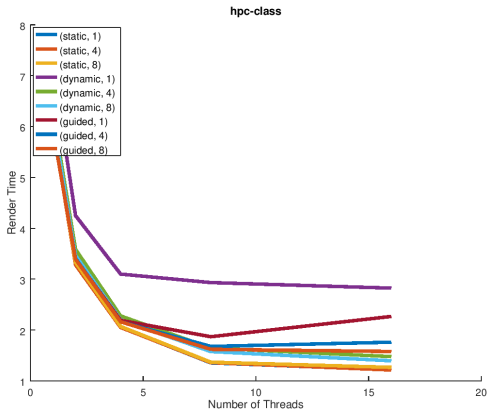
Memory will be used mostly by textures and models which need to be stored in memory for the entire duration of the rendering process. Below is a table which compares peak memory usage against the number of threads used to render 'ironman.obj'. Memory was profiled using heaptrack.

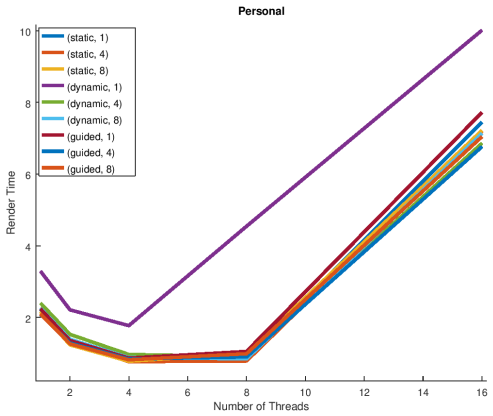
num_threads	peak memory usage
1	131.1 MB
2	131.1 MB
4	131.1 MB
8	131.1 MB
16	131.1 MB

This is exactly the behavior we would expect, since I am only using threads, all memory is shared between processes. All allocations occur outside of threads and threads only perform computations to write to their thread specific location in shared memory. Despite this, memory usage is still quite large, but again this is to be expected do to the nature of the large image and model files needed to render the output.

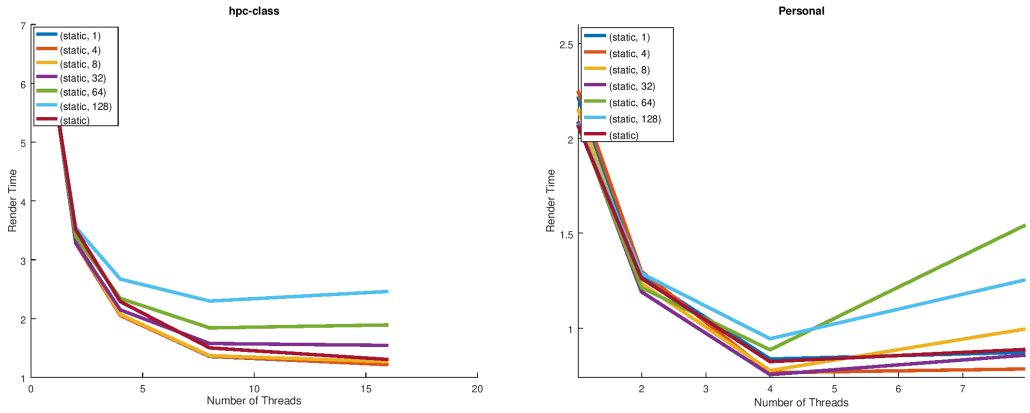
OpenMP Scheduling

Using Method 3, the following graphs analyze different OpenMP scheduling methods and chunk sizes. With the exception of (dynamic, 1) performing notably poor, 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

Here we will again be looking at 'models/ironman.obj' being rendered at 3840×2160. The table below compares the speedup and efficiency for our final method. From these results we can see that so long as the processor has threads to spare, we get an increase in speedup with an increase in threads. However with each added thread we do take a dip in efficiency. Regardless, on the given systems, having a $4.5\times$ peak potential speedup is still quite significant.

thread_count	render time	speedup	efficiency
hpc-class			
1	5.931814	1.0000	100%
2	3.498120	1.6957	84.785%
4	2.295993	2.0040	50.100%
8	1.504625	3.9424	49.280%
16	1.302732	4.5534	28.459%
personal			
1	5.989702	1.0000	100%
2	3.455032	1.7336	86.680%
4	2.017855	2.9684	74.210%
8	1.854047	1.2306	15.382%
16	7.986002	0.75003	4.687%

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.*