Problem Set 2

Collaborators: None.

Problem 2-1. [40 points] **Fractal Rendering**

(a) [1 point] What is the depth of the recursion tree for rendering a snowflake of LoD n?

1. $\log n$
2. n
3. $3\,n$
4. $4\,n$

At each level of the tree, the size of the problem is reduced by 1, so the total depth of the tree is n.

(b) [2 points] How many nodes are there in the recursion tree at level i, for $1 \leq i \leq n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

Each node in the recursion tree represents a call to $Snowflake{-}edge$. The tree begins with three calls to $Snowflake - edge$, and at each iteration, each makes 4 recursive calls to $Snowflake - edge$. At each level i of the tree, there are therefore $3 \cdot 4^i$ nodes.

(c) [1 point] What is the asymptotic rendering time (triangle count) for a node in the recursion tree at level i, for $0 \leq i < n$?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

A single triangle is rendered at each node, so the triangle count is constant, or $\theta(1)$.

(d) [1 point] What is the asymptotic rendering time (triangle count) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

Combining the previous two answers, each level has $3 \cdot 4^i$ nodes, each with a single triangle, for a total triangle count that grows as $\theta(4^i)$.

(e) [2 points] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 3D hardware-accelerated rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

The total cost across all levels is $\sum_{i=0}^{n} 4^i = \theta(4^n)$.

Second, when using 2D hardware-accelerated rendering, the surfaces' outlines are broken down into open or closed paths (list of connected line segments). For example, our snowflake is one closed path composed of straight lines. The CPU compiles the list of coordinates in each path to be drawn, and sends it to the GPU, which renders the final image. This approach is also used for talking to high-end toys such as laser cutters and plotters.

(f) [1 point] What is the depth of the recursion tree for rendering a snowflake of LoD n using 2D hardware-accelerated rendering?

1. $\log n$
2. n
3. $3\,n$
4. $4\,n$

The subproblem size is still reduced by one at each level, so the height is still n.

(g) [1 point] How many nodes are there in the recursion tree at level i, for $1 \le i \le n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

Again, the shape of tree has not changed, so this answer is still $3 \cdot 4^i$.

(h) [1 point] What is the asymptotic rendering time (line segment count) for a node in the recursion tree at level i, for $0 \le i < n$?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

Lines are not rendered until after all triangles has been computed, so node costs at intermediate levels is 0.

(i) [1 point] What is the asymptotic rendering time (line segment count) for a node in the last level n of the recursion tree?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^n)$
4. $\Theta(\frac{1}{3}^n)$

The number of triangles, and hence the number of line segments, is $\theta(1)$.

(j) [1 point] What is the asymptotic rendering time (line segment count) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

The cost at each node is 0, so the total cost is 0.

(k) [1 point] What is the asymptotic rendering time (line segment count) at the last level n in the recursion tree?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

There are $\theta(4^n)$ nodes that cost $\theta(1)$ each, for a total cost of $\theta(4^n)$.

(l) [1 point] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 2D hardware-accelerated rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

All cost is concentrated in the final level, so the total cost is $\theta(4^n)$.

Third, in 2D rendering without a hardware accelerator (also called software rendering), the CPU compiles a list of line segments for each path like in the previous part, but then it is also responsible for "rasterizing" each line segment. Rasterizing takes the coordinates of the segment's endpoints and computes the coordinates of all the pixels that lie on the line segment. Changing the colors of these pixels effectively draws the line segment on the display. We know an algorithm to rasterize a line segment in time proportional to the length of the segment. It is easy to see that this algorithm is optimal, because the number of pixels on the segment is proportional to the segment's length.

Throughout this problem, assume that all line segments have length at least one pixel, so that the cost of rasterizing is greater than the cost of compiling the line segments.

It might be interesting to note that the cost of 2D software rendering is proportional to the total length of the path, which is also the power required to cut the path with a laser cutter, or the amount of ink needed to print the path on paper.

(m) [1 point] What is the depth of the recursion tree for rendering a snowflake of LoD n?

1. $\log n$
2. n
3. $3\,n$
4. $4\,n$

The recursion tree does not change, so height is still n.

(n) [1 point] How many nodes are there in the recursion tree at level i, for $1 \leq i \leq n$?

1. 3^i
2. 4^i
3. 4^{i+1}
4. $3 \cdot 4^i$

Number of nodes at level i is still $3 \cdot 4^i$.

(o) [1 point] What is the asymptotic rendering time (line segment length) for a node in the recursion tree at level i, for $0 \leq i < n$? Assume that the sides of the initial triangle have length 1.

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^i)$
4. $\Theta(\frac{1}{3}^i)$

Line segments are not rendered until the last level, so rendering time is 0 for $i < n$.

(p) [1 point] What is the asymptotic rendering time (line segment length) for a node in the last level n of the recursion tree?

1. 0
2. $\Theta(1)$
3. $\Theta(\frac{1}{9}^n)$
4. $\Theta(\frac{1}{3}^n)$

Line segments are divided by three at each level, but the number of triangles, and hence segments, is constant, so the total segment length at each node is $\theta(\frac{1}{3}^i)$.

(q) [1 point] What is the asymptotic rendering time (line segment length) at each level i of the recursion tree, for $0 \leq i < n$?

1. 0
2. $\Theta(\frac{4}{9}^i)$
3. $\Theta(3^i)$
4. $\Theta(4^i)$

Nodes are not rendered until final level, so rendering time is 0 for $i < n$.

(r) [1 point] What is the asymptotic rendering time (line segment length) at the last level n in the recursion tree?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

Number of nodes grow by $\theta(4^i)$, and line length per node grows by $\theta(\frac{1}{3}^i)$, so total rendering time grows as $\theta(\frac{4}{3}^n)$.

(s) [1 point] What is the total asymptotic cost for the CPU, when rendering a snowflake with LoD n using 2D software (not hardware-accelerated) rendering?

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

All costs are incurred at the final level, so total costs are $\theta(\frac{4}{3}^n)$.

The fourth and last case we consider is 3D rendering without hardware acceleration. In this case, the CPU compiles a list of triangles, and then rasterizes each triangle. We know an algorithm to rasterize a triangle that runs in time proportional to the triangle's surface area. This algorithm is optimal, because the number of pixels inside a triangle is proportional to the triangle's area. For the purpose of this problem, you can assume that the area of a triangle with side length l is $\Theta(l^2)$. We also assume that the cost of rasterizing is greater than the cost of compiling the line segments.

(t) [4 points] What is the total asymptotic cost of rendering a snowflake with LoD n? Assume that initial triangle's side length is 1.

1. $\Theta(1)$
2. $\Theta(n)$
3. $\Theta(\frac{4}{3}^n)$
4. $\Theta(4^n)$

The total area of the snowflake never exceeds the area of the original triangle, so total asymptotic cost is $\theta(1)$.

(u) [15 points] Write a succinct proof for your answer using the recursion-tree method.

Total rendering time grows proportional to the total area of all triangles at all levels. At level i, there are $3 \cdot 4^i$ triangles, and each triangle has area $\frac{1}{3}^{2i} = \frac{1}{9}^i$. The total area over all levels is therefore

$$3 \cdot \sum_{i=0}^{n} (\frac{4}{9}^i) < 3 \cdot \sum_{i=0}^{\infty} (\frac{1}{2}^i) = 1$$

So the total asymptotic rendering time is $\theta(1)$.

Problem 2-2. [60 points] **Digital Circuit Simulation**

Your 6.006 skills landed you a nice internship at the chip manufacturer AMDtel. Their hardware verification team has been complaining that their circuit simulator is slow, and your manager decided that your algorithmic chops make you the perfect candidate for optimizing the simulator.

A ***combinational circuit*** is made up of ***gates***, which are devices that take Boolean (True / 1 and False / 0) input signals, and output a signal that is a function of the input signals. Gates take some time to compute their functions, so a gate's output at time τ reflects the gate's inputs at time $\tau - \delta$, where δ is the gate's delay. For the purposes of this simulator, a gate's output transitions between 0 and 1 instantly. Gates' output terminals are connected to other gates' inputs terminals by ***wires*** that propagate the signal instantly without altering it.

For example, a 2-input XOR gate with inputs A and B (Figure **??**) with a 2 nanosecond (ns) delay works as follows:

The circuit simulator takes an input file that describes a circuit layout, including gates' delays, probes (indicating the gates that we want to monitor the output), and external inputs. It then simulates the transitions at the output terminals of all the gates as time progresses. It also outputs transitions at the probed gates in the order of the timing of those transitions.

This problem will walk you through the best known approach for fixing performance issues in a system. You will profile the code, find the performance bottleneck, understand the reason behind it, and remove the bottleneck by optimizing the code.

To start working with AMDtel's circuit simulation source code, download and unpack the problem set's `.zip` archive, and go to the `circuit/` directory.

The circuit simulator is in `circuit.py`. The AMDtel engineers pointed out that the simulation input in `tests/5devadas13.in` takes too long to run. We have also provided an automated test suite at `test-circuit.py`, together with other simulation inputs. You can ignore these files until you get to the last part of the problem set.

(a) [8 points] Run the code under the python profiler with the command below, and identify the method that takes up most of the CPU time. If two methods have similar CPU usage times, ignore the simpler one.

```
python -m cProfile -s time circuit.py < tests/5devadas13.in
```

Warning: the command above can take 15-30 minutes to complete, and bring the CPU usage to 100% on one of your cores. Plan accordingly.

What is the name of the method with the highest CPU usage?

Answer: method_ name

(b) [6 points] How many times is the method called?

Answer: 0

(c) [8 points] The class containing the troublesome method is implementing a familiar data structure. What is the tightest asymptotic bound for the worst-case running time of the method that contains the bottleneck? Express your answer in terms of n, the number of elements in the data structure.

1. $O(1)$.
2. $O(\log n)$.
3. $O(n)$.
4. $O(n \log n)$.
5. $O(n \log^2 n)$.
6. $O(n^2)$.

Answer: 0

(d) [8 points] If the data structure were implemented using the most efficient method we learned in class, what would be the tightest asymptotic bound for the worst-case running time of the method discussed in the questions above?

1. $O(1)$.
2. $O(\log n)$.
3. $O(n)$.
4. $O(n \log n)$.
5. $O(n \log^2 n)$.
6. $O(n^2)$.

Answer: 0

(e) [30 points] Rewrite the data structure class using the most efficient method we learned in class. **Please note that you are not allowed to import any additional Python libraries and our test will check this.**

We have provided a few tests to help you check your code's correctness and speed. The test cases are in the `tests/` directory. `tests/README.txt` explains the syntax of the simulator input files. You can use the following command to run all the tests.

```
python circuit_test.py
```

To work on a single test case, run the simulator on the test case with the following command.

```
python circuit.py < tests/1gate.in > out
```

Then compare your output with the correct output for the test case.

```
diff out tests/1gate.gold
```

For Windows, use `fc` to compare files.

```
fc out tests/1gate.gold
```

We have implemented a visualizer for your output, to help you debug your code. To use the visualizer, first produce a simulation trace.

```
TRACE=jsonp python circuit.py < tests/1gate.in > circuit.jsonp
```

On Windows, use the following command instead.

```
circuit_jsonp.bat < tests/1gate.in > circuit.jsonp
```

Then use Google Chrome to open `visualizer/bin/visualizer.html`

We recommend using the small test cases numbered 1 through 4 to check your implementation's correctness, and then use test case 5 to check the code's speed.

When your code passes all tests, and runs reasonably fast (the tests should complete in less than 30 seconds on any reasonably recent computer), upload your modified `circuit.py` to the course submission site.

Answer: Please upload `circuit.py` with your modifications separately.