Day zero: Legenda

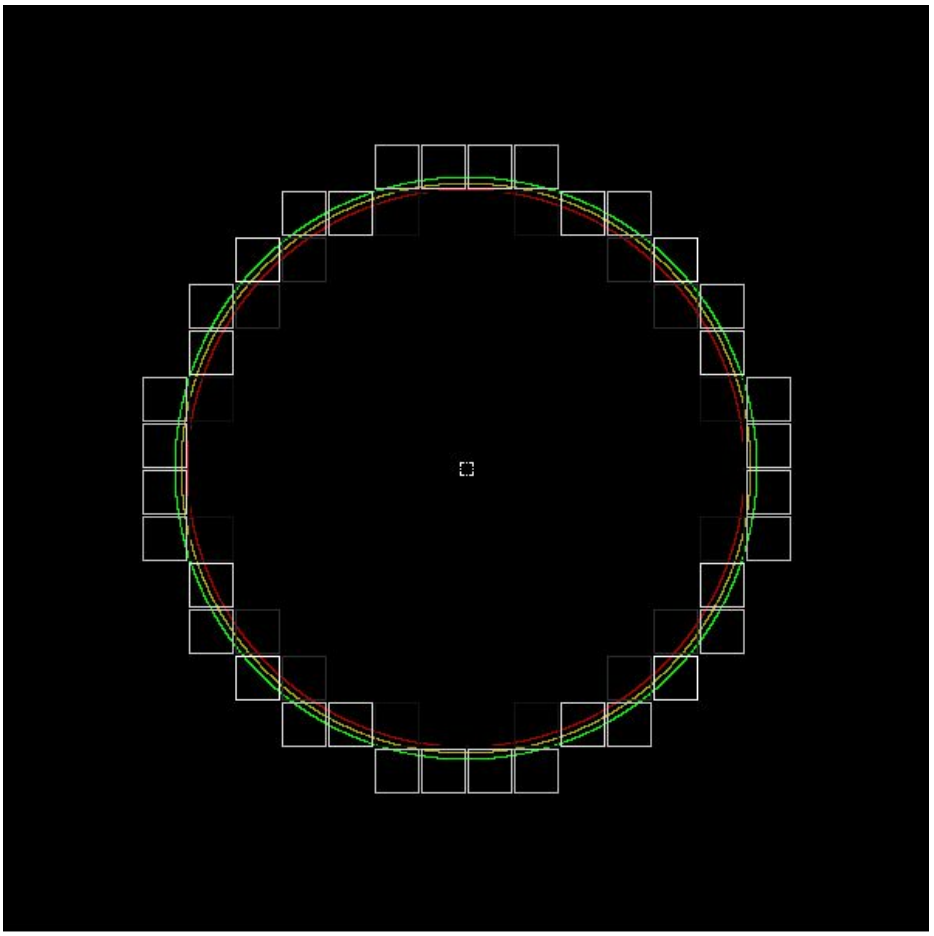
“Orientation and understanding is the first step towards any adventure. Or math question.”

This chapter contains a few definitions used throughout this report.

Library: The Swarmz library is abbreviated to ‘the library’ or just ‘library’.

Boid: A boid is a point in space that is part of the simulation. In literature, a boid is considered to be a bird.[1]

Benchmark: Throughout the document we’ll use the ‘Circle test’ as our benchmarking. This provides the most extremely scenario: when all boids are on top of one another, and therefore should keep track of one another. It looks like the following:



The benchmark. Take note that all boids are moving towards the center. When the boids go outwards again, they eventually split up into different swarms.

[1] todo: find boid thing paper

Parameters: The library contains various parameters that influence the performance. We use the default parameters provided by the library. These are:

- Alignment weight = 1.0
- Cohesion weight = 1.0
- Steering weight = 0.1
- Separation weight = 1.0

- Perception radius = 30.0
- Blindspot angle deg = 20.0

- Max acceleration = 10.0
- Max velocity = 20.0

Performance: The benchmark is used when measuring the performance. We measure at the start (second frame) and when all the boids are in the middle. We visualise this with:

```
(x -> y)
```

Where x is the value at the start, while y is the value when all the boids are in the middle. All performance measurements are in milliseconds per frame (ms). All performance measurements are taken with 20000 boids, unless mentioned otherwise. The frametime for the original is very slow, which would cause benchmarking to take 10 minutes. This is why in most steps, we only measure x, but for the end we also measure y.

Day one: checking out the original

“Our first baby steps.”

Git hash: 6a344136a9b6757a3a2432a8a0c906363834194a

The original library shows significant slowdowns throughout the benchmark when using merely 20000 boids. The original performance `(601 -> 21,480)` is rather slow. Profiling shows various reasons:

Top Hotspots

This section lists the most active functions in your application. Optimi

Function	Module	CPU Time
sw::Swarm::getNearbyBoids	Tmpl8_2018-01.exe	13.304s
func@0x18002f6ac	ucrtbase.dll	9.151s
func@0x180032a18	ucrtbase.dll	3.030s
func@0x18002f438	ucrtbase.dll	2.642s
sw::Swarm::updateBoid	Tmpl8_2018-01.exe	2.329s
[Others]		4.004s

**N/A is applied to non-summable metrics.*

Both the func@... hotspots are from the *ucrtbase.dll* library.

```
270             for (int x = 0; x < 4; x++) {
271                 for (int y = 0; y < 4; y++) {
272                     for (int z = 0; z < 4; z++) {
273                         checkVoxelForBoids(b, result, voxelPos);      38.5%   13.265
274                         voxelPos.Z++;                                 0.0%    0.009
275                     }
276                     voxelPos.Z -= 4;
277                     voxelPos.Y++;                                     0.0%    0.010
278                 }
```

When looking at the hotspot, the entire checkVoxelForBoids function is inlined. This prevents us from looking at where the actual costs are. Temporarily disabling inlining provides more insight. We are aware that disabling inlining hurts performance. Profiling without inlining provides more insight in what is consuming the CPU cycles.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
func@0x180032a18	ucrtbase.dll	11.639s
func@0x18002f6ac	ucrtbase.dll	8.711s
sw::Vec3::AngleTo	Tmpl8_2018-01.exe	6.794s
std::vector<sw::NearbyBoid,std::allocator<sw::NearbyBoid> >::_Emplace_reallocate<sw::NearbyBoid const &>	Tmpl8_2018-01.exe	6.588s
sw::Vec3::Length	Tmpl8_2018-01.exe	3.702s
[Others]		23.177s

After this, ucrtbase.dll costs the most time. This dll contains the unordered map functionality, which is implemented with a linked list. Since linked list is known to be slow, this should be rewritten to a new data structure.

In this case, it seems that specific inline functions that were called inside the checkVoxelForBoids were the most costly in this function.

- **AngleTo has a static cast and acos function**, which is quite slow.

```
float AngleTo(const Vec3 &v) const {                                  1.0%    0.603s
    float l1 = Length();                                              0.0%    0.010s
    float l2 = v.Length();                                            0.5%    0.315s
    if (l1 == 0 || l2 == 0) {                                         0.9%    0.525s
        return 0;
    }
    return static_cast<float>(std::acos(DotProduct(v) / (l1 * l2)) * 360 / PI2);   7.4%   4.512s
}                                                                     1.4%    0.828s
```

- **vector<NearbyBoid> needs to reallocate** quite often (see Top hotspots)
- The **Vec3 Length** calculation is also quite expensive.

```
float Length() const {                                                0.4%    0.246s
    return std::sqrt(std::pow(X, 2) + std::pow(Y, 2) + std::pow(Z, 2));   5.2%   3.132s
}                                                                     0.5%    0.325s
```

Since this works with std::pow and sqrt, that is not too surprising.

One of the calls to the *ucrtbase.dll* library is done through an unordered map that is used to map the boids into cells. These calls turn out to be expensive and take the majority of the time of the *getNearbyBoids* function. This is a potential high level optimisation.

The calls to *AngleTo* and *Length* are done frequently and these include expensive square root and cosine instructions. The cosine is the other expensive *ucrtbase.dll* library call. This is a potential low level optimisation.

The next step would be split in two: the first is to work on the Grid implementation to remove the unordered map . To work efficiently, we branched so the other could work on the AngleTo step.

Day two: Grid

“They told us to not think in boxes. They were wrong. So wrong.”

Git hash: 84689e2ca6e5e29577bf7c97200ce6e301b80c0f

Our initial approach is on the high level - removing the unordered map and replacing it with a grid structure. We did this because we knew we would have to parallelize the structure eventually and since this was a bottleneck, we decided to rewrite this straight away. The performance (`254 -> 3808`) is significantly improved, but the grid has a limited capacity in each cell. *Some sets of boids do not fit entirely in a cell, causing deviating behavior of the boids and yielding* ***a fake feeling of improved performance****.*

Profiling provides us with the following results. Take note that inlining is again disabled for identifying the actual Hotspots.

Hotspots	*Microarchitecture*
func@0x180030328 AngleTo (...) QueryGrid (...) Length (...)	*42% retiring* *26% memory bound* *24% core bound*

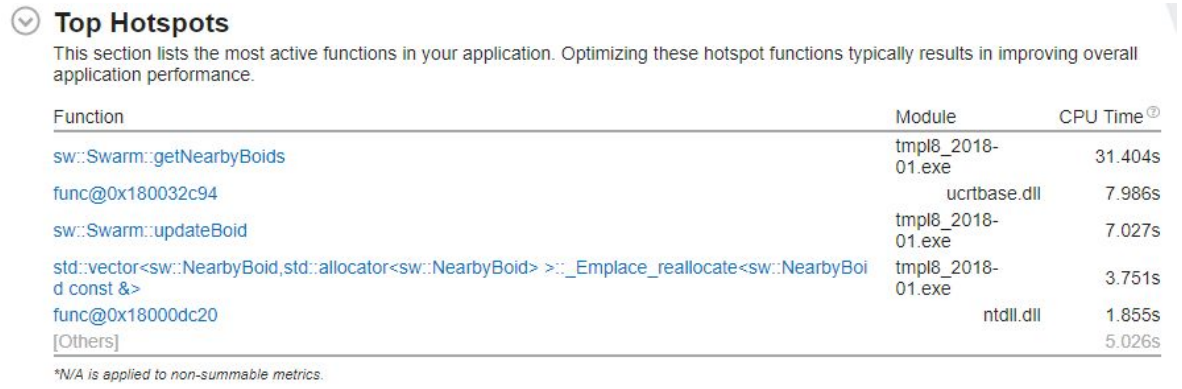
One of the *ucrtsbase.dll* calls has disappeared and has been replaced by the QueryGrid call which is positioned lower than before. This is good. On the other hand, we’re significantly more memory bound than before, which indicates that we’re having random reads and / or writes through memory, causing significant cache misses all the way down to DRAM. This makes sense - the Grid is one large scattered memory access. More so than before, were unoccupied cells would not be visited.
The func@... is currently unknown, but it will turn out to be gone after AngleTo was adjusted. Therefore this was probably the acos function.

Day twopointfive: AngleTo / Length

“We’ll approach it from another angle.”

Git hash: cb624b6432147e2b1b9b89c79f28779920662f7c

Our second approach is through low level optimisations. The performance `(340 -> y)` increased here too. Take note that this was on a separate branch, the grid changes were on another branch. We chose to target the *AngleTo* and *Length* function
Since the grid was not fully working due to ignoring a part of the boids if it doesn't fit in a cell, we decided to split tasks and continue with the AngleTo and Length methods. The pow(…, 2) methods were replaced by multiplying the variable by itself and sqrt() was changed to sqrtf() to prevent a cast. Thereby an AngleToNorm was made, which takes vectors that are already normalized, to prevent duplicate length calculations.
The acos calculation is now performed with a lookup table, which automatically gives floats so it also prevents a cast.



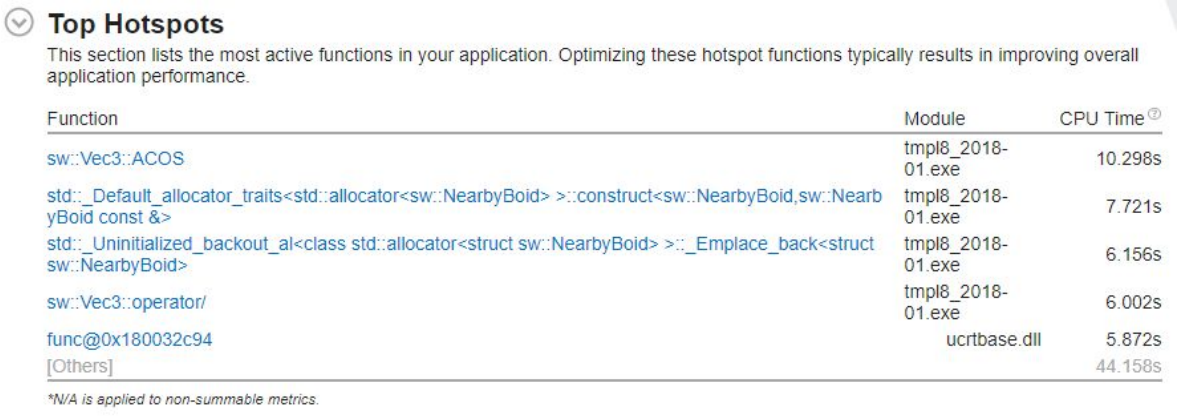
Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
sw::Swarm::getNearbyBoids	tmpl8_2018-01.exe	31.404s
func@0x180032c94	ucrtbase.dll	7.986s
sw::Swarm::updateBoid	tmpl8_2018-01.exe	7.027s
std::vector<sw::NearbyBoid,std::allocator<sw::NearbyBoid> >::_Emplace_reallocate<sw::NearbyBoid const &>	tmpl8_2018-01.exe	3.751s
func@0x18000dc20	ntdll.dll	1.855s
[Others]		5.026s

**N/A is applied to non-summable metrics.*

Using inlining, the AngleTo Function could be split up further, which showed a bottleneck in mostly ACOS, which is the lookup table.
Without inlining:



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
sw::Vec3::ACOS	tmpl8_2018-01.exe	10.298s
std::_Default_allocator_traits<std::allocator<sw::NearbyBoid> >::construct<sw::NearbyBoid,sw::NearbyBoid const &>	tmpl8_2018-01.exe	7.721s
std::_Uninitialized_backout_al<class std::allocator<struct sw::NearbyBoid> >::_Emplace_back<struct sw::NearbyBoid>	tmpl8_2018-01.exe	6.156s
sw::Vec3::operator/	tmpl8_2018-01.exe	6.002s
func@0x180032c94	ucrtbase.dll	5.872s
[Others]		44.158s

**N/A is applied to non-summable metrics.*

`inline float ACOS( float x )`		
`{`	0.3%	77.727ms
`assert( x >= -1.1 && x <= 1.1 ); //there c`		
`x = std::min( x, 1.0f );`	0.3%	96.241ms
`x = std::max( x, -1.0f );`	1.7%	482.208ms
`int i = ( x + 1 ) / indexToAcosRange;`	3.5%	993.043ms
`return acosTable[i];`	2.2%	621.424ms
`}`	1.7%	475.133ms

Currently we can see no way to optimize this without parallel processing.

The Nearbyboid bottlenecks would be gone as soon as we merged the buckets branch.
With regard to the Microarchitecture, the Divider and Port utilizations are potential targets.
With regard to the hotspots, the vector allocation is a potential target.
Since the bucket branch was done, and

Day fourhundredandseventysix: SoA & Buckets

“Let’s try a structured approach. With buckets.”

Git hash: c92ba3a533adb5f61a3ec3f2aff4756aefddba1e

Our next approach was to prepare the intermediate structure for SiMD operations. In general, implementing the SoA structure would be more cache line friendly and therefore should be faster on its own due to linear memory reads / writes.

The SoA structure is implemented only internally to preserve the public API. Each cell contains the SoA data layout, and when a boid should be present in the given cell the boid is then stored into the SoA data layout. All the data in the grid is read-only.

Our other target was making the grid consistent with the original implementation. We did this through a bucket system, a grid cell can ask for one or more buckets depending on the demands of the grid cell. This allows for ‘dynamic memory allocation’ that is all already allocated and GPU friendly. All buckets used the SoA data layout.

The performance (`207 -> y`) is improved nicely. This is because the actual number of boids are now properly represented. Compared to the original, this is almost a two times speedup.
Profiled with inlining on:

Function	Module	CPU Time
sw::Grid::QueryGrid	tmpl8_2018-01.exe	33.666s
sw::Swarm::UpdateAcceleration	tmpl8_2018-01.exe	0.184s
sw::Swarm::accelerateByForce	tmpl8_2018-01.exe	0.132s
func@0x180cff120	nvd3dumx.dll	0.079s
func@0x1800012d0	vcruntime140.dll	0.052s
[Others]		0.945s

**N/A is applied to non-summable metrics.*

The checkVoxelForBoids function has a new name in the code: QueryGrid. This means the hotspot is still the same.

`Vec3 distanceVec = p2 - p1;`	8.0%	0.854s
`float distance = distanceVec.Length();`	2.3%	0.245s
`// check if they are the same or not ( todo: this is broken at this point)`		
`if ( distance > 0.00001f )`	4.5%	0.480s
`{`		
`    // check if the distance is nearby enough`		
`    if ( distance <= PerceptionRadius )`	0.5%	0.049s
`    {`		
`        Vec3 distanceVecNorm = distanceVec / distance;`		
`        Vec3 bNegVelocity = b.Velocity.Negative();`	1.8%	0.195s
`        float bNegVelocityLength = bNegVelocity.Length();`	2.5%	0.270s
`        float blindAngle = 0;`	0.2%	0.019s
`        if ( bNegVelocityLength > 0.000001f && distance > 0.00001f )`	1.6%	0.175s
`        {`		
`            Vec3 bNegVelocityNorm = bNegVelocity / bNegVelocityLength;`	3.8%	0.408s
`            blindAngle = bNegVelocityNorm.AngleToNorm( distanceVecNorm );`	13.9%	1.487s
`        }`		
`        // check if we can 'see it'`		
`        if ( BlindspotAngleDeg <= blindAngle || bNegVelocityLength == 0 )`	4.9%	0.531s
`        {`		
`            NearbyBoid nb;`		
`            // was: nb.boid = &target`		
`            nb.boid = target;`	0.7%	0.078s
`            nb.distance = distance;`		
`            nb.direction = distanceVec;`	6.1%	0.656s
`            out.emplace_back( nb ); //TODO dit is vaag, moet met mov toch?`	19.4%	2.085s
`        }`		

Profiled without inlining:

Function	Module	CPU Time
sw::Grid::QueryGrid	tmpl8_2018-01.exe	12.852s
sw::Vec3::ACOS	tmpl8_2018-01.exe	12.697s
Tmpl8::vec3::length	tmpl8_2018-01.exe	4.252s
sw::Vec3::operator/	tmpl8_2018-01.exe	3.325s
Tmpl8::vec3::vec3	tmpl8_2018-01.exe	2.744s
[Others]		15.730s

*N/A is applied to non-summable metrics.

emplace_back in QueryGrid costs some time, but since our next step would be writing the array of structures (AOS) to a structure of arrays (SOA) to implement SIMD and GPGPU, the vector would disappear and so would the instruction.

The p2 - p1 line also costs some time, but we can see no way to further optimize this. This means the next step would be to rewrite the AOS to SOA.

Day fourhundredandseventyseven: interleaving & loophoisting

“Let’s just do it once, shall we.”

Git hash: 356ee58d7ee6896876b792b4f3e09a12bc743812

We discovered a few constants in our expensive functions, which we excluded outside the for loop. An example of this was the length of the velocity of the boid we were doing the computations for.

Through interleaving we tried reducing the bubbles. Some intermediate results were saved, reducing the number of direct dependencies, decreasing the bubble in the stream. This was especially noticeable with the square root operations.

The performance (`114 -> 3115`) nominally increased after everything was completed.

Function	Module	CPU Time
sw::Grid::QueryGrid	tmpl8_2018-01.exe	184.393s
BucketPool::GetBucket	tmpl8_2018-01.exe	0.695s
sw::Swarm::UpdateAcceleration	tmpl8_2018-01.exe	0.371s
sw::Swarm::accelerateByForce	tmpl8_2018-01.exe	0.270s
func@0x180cff120	nvd3dumx.dll	0.203s
[Others]		1.987s

**N/A is applied to non-summable metrics.*

At this point we are core-bound, VTune advised us to vectorise our code. So that is what we did next.

Day octo-fourhundredandseventyeight: SiMD and battling the number of retirements

“An attempt to being more productive.”

Git hash: 37f370f415ca5af91e56973c63c7ba3580f2f15c

On top of that, we introduced SiMD instructions that clearly mapped over our SoA data layout. The expected speedup was minimal, merely a 1.5 time speedup. This was partially caused by the bucket system introduced earlier causing random read / write accesses, and partially due to bubbles in the pipeline.

We’ve fixed that soon after through reducing the number of dependencies between instructions. Via storing intermediate result into temporary buffers the CPU was able to process the data significantly faster and being able to hide latencies more properly.

The performance (`60 -> 1224`) is not as significant as we anticipated, it is a rough four times increase but we anticipated an eight times increase due to AVX instructions.

Function	Module	CPU Time
sw::Grid::QueryGrid	tmpl8_2018-01.exe	27.745s
sw::Swarm::accelerateByForce	tmpl8_2018-01.exe	0.361s
memset	vcruntime140.dll	0.358s
sw::Swarm::UpdateAcceleration	tmpl8_2018-01.exe	0.334s
BucketPool::GetBucket	tmpl8_2018-01.exe	0.201s
[Others]		1.688s

**N/A is applied to non-summable metrics.*

According to VTune the theoretical maximum of CPI is 0.25, to which we got pretty close at this point. Our next target is using more cores in general.

Day eight: Multithreading

“I think my laptop just did a sonic boom.”

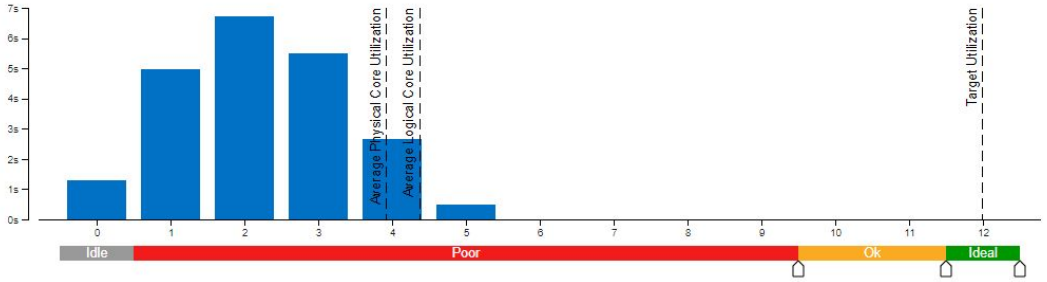
Git hash: 2f80bf8edcbf3a6ae7b2848630dbdb00585f2217

The performance (`11 -> 310`) increased significantly in comparison to the previous version.

The effectiveness of the multi threading is not optimal yet.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



The hotspots suggest that we may have locking issues.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
sw::Grid::QueryGrid	tmpl8_2018-01.exe	72.060s
RtlAcquireSRWLockShared	ntdll.dll	7.718s
RtlReleaseSRWLockShared	ntdll.dll	7.639s
memset	vcruntime140.dll	0.876s
sw::Swarm::UpdateAcceleration	tmpl8_2018-01.exe	0.639s
[Others]		5.906s

**N/A is applied to non-summable metrics.*

At this point in time we do not have enough time to solve this optimization issue. Therefore our final speed up is:
original(601 -> 21480) to our version (11 -> 310) = (51x -> 69x). This is quite a speed up, allowing for 60 fps or more when the boids are properly spread out. Especially when there are *only* 5000 boids the threshold of 60 fps is reached with ease through a staggering framerate of (2 -> 5). Due to the quadratic nature when all boids are centered, the difference is less significant.

The difference of acceleration at the edge and in the center of the cirkel can be explained due to the type of optimisations. For example, the more boids the more SiMD will be able to shine.

Sadly we did not have enough time to optimize the multithreading further and to implement GPGPU. We will still do GPGPU after the deadline to learn it, however that will of course not be a part of our grade. Hurrr Hurr.