Technical Design

[Taken by *Daniel Dexter-Taylor*]

Conway's Game of Life 3D

0. Contents

0.1 With Thanks

Adam Russell
Eleanor Cullen

1. Design Problems

1.1 Array Caching

When accessing an array sequentially this proves to be relatively fast, with only the occasional cache miss as chunks of the array are cached when element lower in the array are loaded from memory. The problem occurs, however, when access to the array is not sequential, as this will cause a large amount of cache misses.

1.2 Conway's Game of Life in 3D

The best way to program Conway's Game of Life (GOL) is to use an array of Booleans and access them directly that way; however this means when checking an element's neighbour's access to the array is not sequential. Each check on each element would have to access the following elements, where *s* is the dimensions of the grid and *n* is the element number:

$n - 1,\ n + 1,\ n - s,\ n + s$

This problem is exacerbated when you add an extra dimension to GOL, like so:

$n - 1,\ n + 1,\ n - s,\ n + s,\ n - s^2,\ n + s^2$

(See appendix 5.1)

Array accessing like this would result in approximately a five out seven miss rate.

1.3 Object Orientation

For ease of programming, in style and repetition of code, I elected to break down my code into three main functions: initialise, update and render. I also broke down the source files into three files: main, cube data and front end. The reason for this was to separate the static data and functions required for rendering, from the static data and functions required for game logic.

Almost all the functions were set to be inlined, however the compiler efficiency overrode this anyway and decided itself what should be inlined. As a result the smaller functions, such as the accessors and mutators, were inlined. This wasn't a problem, rather more what I was intending, because the functions themselves were very small they were not worth keeping as separate functions for the sake of space, as well as the overhead and time it takes to jump to and from them in execution.

2. Design Solutions

2.1 Game of Life States

In order to reduce, or even eliminate, cache misses when accessing the GOL grid array I decided to compact the size of the data into the smallest amount possible by using bit shifting. Originally I had planned to store the current and previous data side-by-side, however after finding this too difficult to get data in and out and archiving, I opted for a struct containing two unsigned shorts (sixteen bits long).

What this meant was a sixteen square array would take up only eight kilobytes of memory, which fits entirely into the PSP's sixteen kilobyte I cache.

2.2 Inline Accessors & Mutators

The quickest and simplest way to use the GOL grid was to abstract access and manipulation through an accessor and mutator. The idea was to loop through the x, y and z coordinates, find the element in the array by multiplying out the z and y coordinates, then pass the accessor/mutator the index of the array and x value.

By inlining these functions it reduced the amount of code I wrote while still executing seamlessly without jumps in the execution.

2.3 Allegrex Built-in Functions

As the entire GOL grid array only took up eight kilobytes I decided to force the CPU to pre-load the entire array before update, by using a built-in function on the Allegrex CPU. Futhermore, I decided to only force this pre-load once at creation of the array and lock it in the cache. After testing this, however, I found that this had a negative impact on the execution speed.

(See appendix 5.2)

	Frame Time (ms)			Frame Sync (ms)		
	Min	**Max**	**Avg.**	**Min**	**Max**	**Avg.**
With	50.021	83.425	50.481	0.625	9.476	3.684
Without	50.029	66.730	50.311	1.812	9.792	5.259

As you can see from the above summary, without pre-loading the array into cache and locking it is more efficient than doing so. The result of this was to change the cache loading to two different places. Firstly, during `CubeData::UpdateCubes()` the entire array is pre-loaded and locked into the I cache, then unlocked at the end of the function. Secondly, in `CubeData::ReleaseRender()` before the third loop is hit the entire line is pre-loaded into the I cache without locking.

(See appendix 5.3)

2nd With	50.009	83.398	50.488	5.295	16.004	6.155

As you can see, although the average frame time is actually worse the frame sync time is better. The only conclusion I can draw by this is that the array is so small that the overhead involved in pre-loading it into the cache is actually detrimental to performance, and it's better to let the architecture deal with caching itself. However, if I hadn't minimised the size of the array then manual caching would most likely had a profound positive impact on performance.

3. Front End

3.1 Cube Example

The front end was mainly based on the *Cube Example* from Sony's PSP examples, this was the main reason for separating the logic and render code into two separate files so that all the code I'd written myself was in one file and all the code from the example, which uses a different style/formatting.

Two static variables, `FrontEnd::disp_list` and `FrontEnd::matrix_stack`, I changed in for optimisation. The matrix stack I reduced from thirty-six to only four, as I only used four matrices at most. Also, I found that when rendering a lot of cubes on screen the PSP would crash; after increasing the disp list by one-hundred times the amount this fixed the crashing.

3.2 Vertex Data

In order to save on space, I removed all the UV and normal data from the vertices, which compacted down each vertex from seven floats to three floats. Although the data was not indexed I chose not to manually index them because the total static data was under the one kilobyte limit, moreover changing it to an indexed array would not give a substantial reduction in execution speed.

3.3 Inline Rendering

Most of the matrix calculations and transformations are done inside the `CubeData::ReleaseRender()` function, however the `FrontEnd::RenderCube()` function is not, because this was based off the *Cube Example*. I decided to inline this, as with most of the smaller functions, to reduce the number of jumps being executed.

The grid that each cube is rendered to rotates in front of the camera, which means each cube uses the same rotation matrix to rotate it around the origin. I found that if I created this matrix before rendering and stored it on the function stack, then pushed it back into the matrix stack each time a cube is rendered this saved on CPU time by creating this matrix once instead of every time a cube is rendered.

3.4 Interpolating Colours

Each cube is coloured different, finding their colour during real-time by interpolating their colour (BGR) based on their positions (ZYX) respectively. I did this by casting the position to a float, normalising it and multiplying by two-hundred and fifty-five. Casting from a float to an integer, then back again, is very expensive on the CPU; however to counteract the time spent casting from a float to an integer I used a built-in function on the Allegrex to round it to a whole number. This saved on CPU time greatly.

4. PSP-GCC

4.1 Compiler Efficiency

For the majority of development I used no optimisation, in order to remove the risk of bugs produced by compiler efficiency and also gain complete control over what the code is doing in order to easily influence code efficiency.

When I did change the level of compiler efficiency, however, I found that anything above *o1* caused the program to crash in a variety of ways. After investigation I found this was because the make file I was using, which was based on the *Fountain Example*, used two additional optimisation flags as well, which caused problems in this compile. Also, I found that *.elf* files crashed shortly after execution. This problem has not been resolved, neither have I found the reason for it.

4.2 Comparing Sizes

In order to reduce the size of executable to fit the four kilobyte constraint, I compared each built-in level of efficiency in the PSP-GCC.

(See Appendix 5.4.1)

As you can see *oS* is the only level which compacts the text size to below four kilobytes, this is because the efficiency flags used in *oS* are based on *o2* but modified to reduce code size without sacrificing efficiency. More interesting though is the word size at after *o3* doesn't change. This pattern continues when we compare the data size for each level.

(See Appendix 5.4.2)

As you can see after *o2* the word size doesn't change, which leads me to the conclusion that, in this particular case, the data is as efficient as it can ever be after only *o2* optimisation. The same can be said about the code, which in this case is as small as it can be after *o3* optimisation.

4.3 Comparing Efficiency

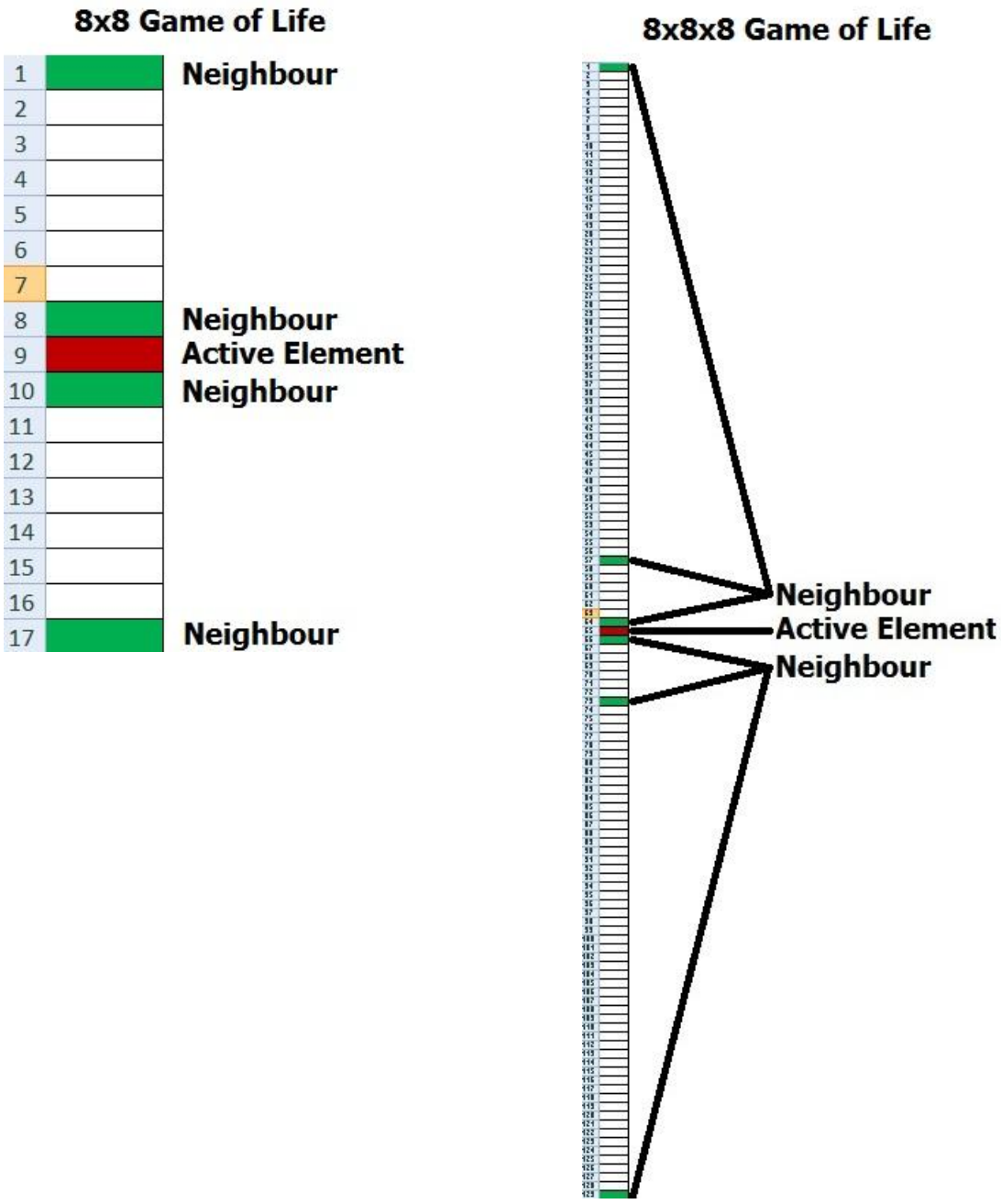
(See Appendix 5.4.3)

Comparing o3, o4 and o5 efficiency on average frame time, each are within 0.005 milliseconds of each other. Furthermore the average frame sync times are within 0.056 milliseconds, both of which are a negligible change attributable to uncontrollable changes. This supports the claim that these levels of efficiency are already at their optimal.

After using any level of compiler efficiency the average frame time is brought under fifty milliseconds, where the average frame time without efficiency was over eighty milliseconds. This is a drop to almost half the frame time. Although using *o3* optimisation gave the best frame time, in this case the change between *oS* and *o3* was negligible since it gave no noticeable lag.

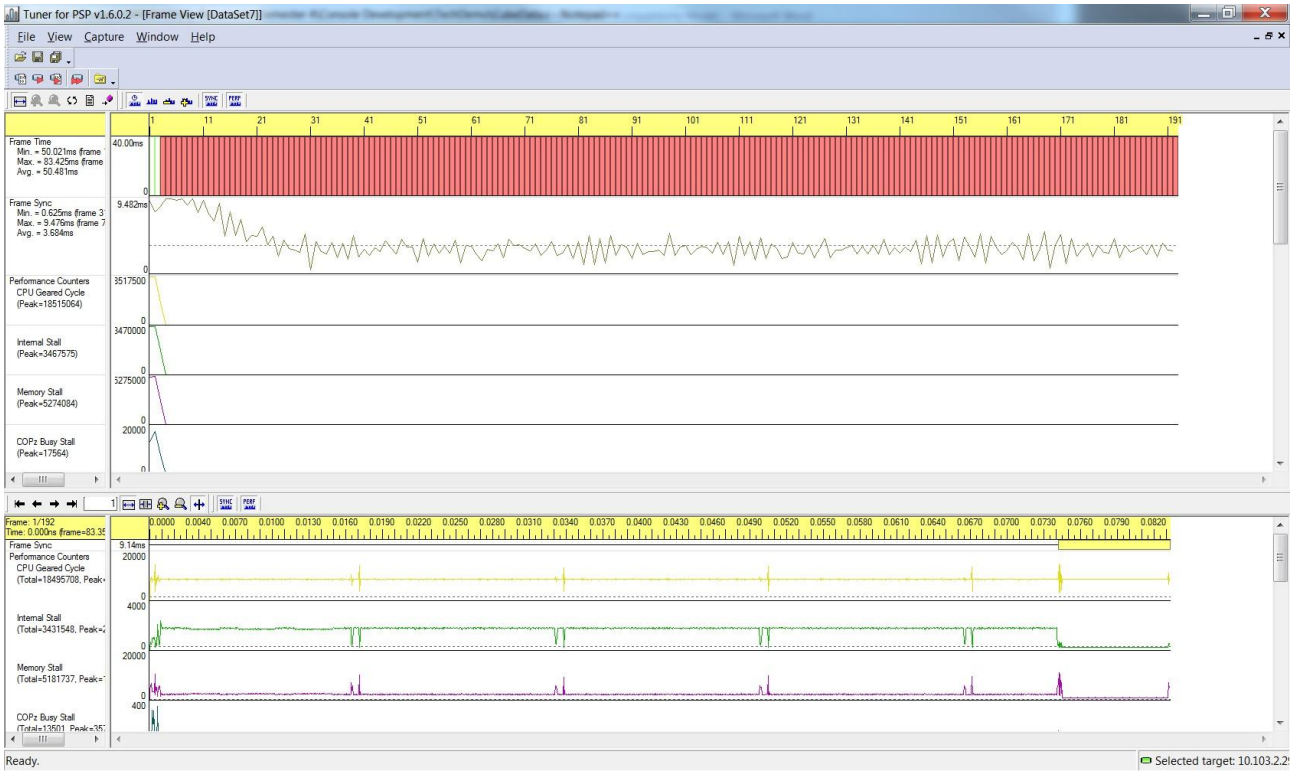
However, that being said *oS* did have the worst average frame sync by approximately four milliseconds. Although this is has no real impact on performance, it's important to note as you would expect a result with lowest average frame time to have the greatest average frame sync.

5.1 Array Accessing

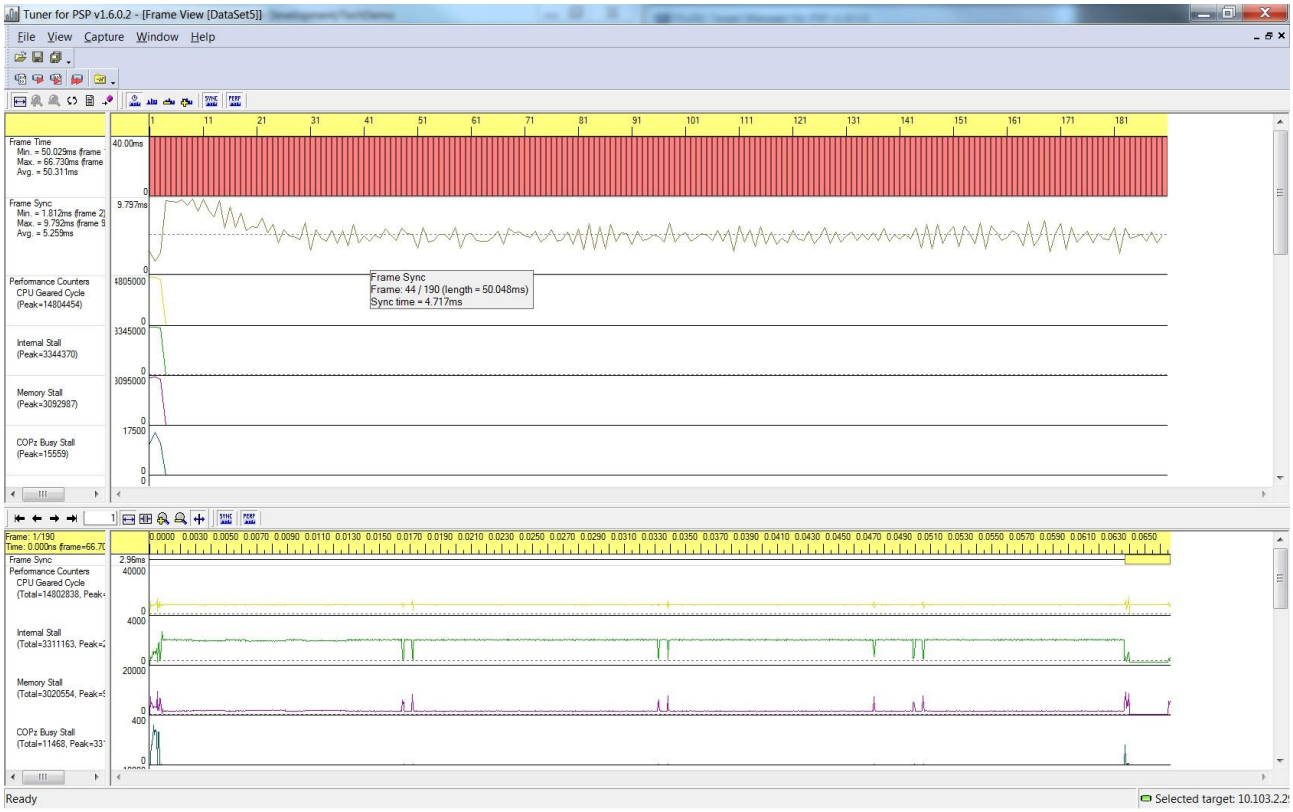


5.2 Tuner Results (Locking Cache)

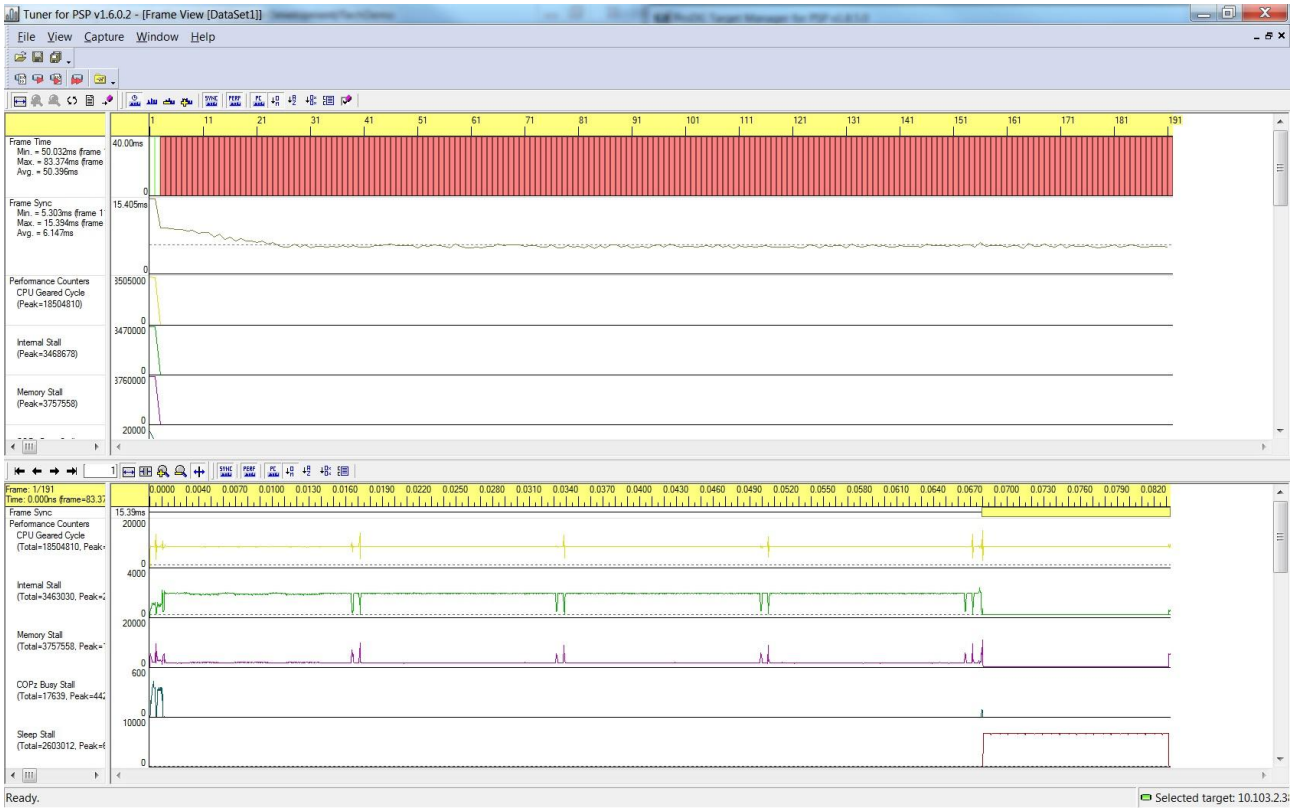
5.2.1 With Cache Pre-Loading



5.2.2 Without Cache Pre-Loading



5.3 Tuner Results (Second Optimisation)



5.4 Compiler Efficiency Levels

5.4.1 Text Sizes

	o0	o1	o2	oS	o3	o4	o5
Main	104	80	80	80	80	80	80
CubeData	6164	4504	4968	3284	4976	4976	4976
FrontEnd	956	688	672	672	672	672	672
Total	**7224**	**5272**	**5720**	**4036**	**5728**	**5728**	**5728**
Spare	**-3128**	**-1176**	**-1624**	**60**	**-1632**	**-1632**	**-1632**

5.4.2 Data Sizes

	o0	o1	o2	oS	o3	o4	o5
Main	448	448	56	56	56	56	56
CubeData	556	560	124	124	124	124	124
FrontEnd	448	448	448	448	448	448	448
Total	**1452**	**1456**	**628**	**628**	**628**	**628**	**628**
Spare	**-428**	**-432**	**396**	**396**	**396**	**396**	**396**

5.4.3 Frame Rates

		o0	o1	o2	oS	o3	o4	o5
Frame Time	**Min**	83.394	16.609	16.632	33.322	16.605	16.636	16.612
	Max	83.440	50.100	50.073	50.074	50.063	50.068	50.051
	Avg.	**83.414**	**46.767**	**36.778**	**48.269**	**33.960**	**33.956**	**33.951**
Frame Sync	**Min**	6.412	1.046	0.234	0.345	0.233	0.279	0.288
	Max	15.044	16.436	16.880	16.207	16.867	16.639	16.846
	Avg.	**8.662**	**9.379**	**5.521**	**13.336**	**3.625**	**3.553**	**3.609**