The Graphics Unit of the INTEL I80860

Ulrich Kursawe

KONTRON ELEKTRONIK GmbH
Research & Development Department
Breslauer Str. 2, D-8057 Eching, FRG

The Intel I80860 is a very powerful RISC processor, designed for applications that require a large amount of floating point and integer calculations. Additionally it supports graphics applications with a Graphics hardware unit. The aim of this article is to investigate, for which application this unit is useful and whether the results obtained by the help of this unit are better as with standard C or assembly implementations of the same algorithm.

1. General Aspects

Although the Intel I80860 is mainly designed as a fast floating point number cruncher, it also features a graphics unit. This Graphics Unit (GU) is useful at the low end of special algorithms for pixel manipulation. Geometric transformations and other manipulations are very well covered by the high-speed floating point and integer unit, which can be accessed simultaneously. Additionally the floating point adder and multiplier can operate in parallel, although this requires hand-optimised assembly routines. This goes as well for routines which want to use the graphics unit effectively. See figure 1.

All compilers use the GU only to transfer data from floating point registers to integer registers and vice versa. So the usage of the GU in shading algorithms requires always that at least part of the program code is assembly code.

The I80860 supports pipelining with special instructions and may therefore be considered as a vector processor. Again the compilers do not use pipelining (with the exception of the PSR vectoriser) and the user who wants to use pipelined operations has to manage the different pipelines and keep track of their actual status himself. This is not trivial because the I80860 has 5 different pipelines with different numbers of stages:

a) Single and Double precision FP–Adder Pipeline with 3 stages
b) Single precision FP–Multiplier Pipeline with 3 stages
c) Double precision FP–Multiplier Pipeline with 2 stages
d) Graphics Unit Pipeline with 1 stage
e) FP–Loader Pipeline with 3 stages.

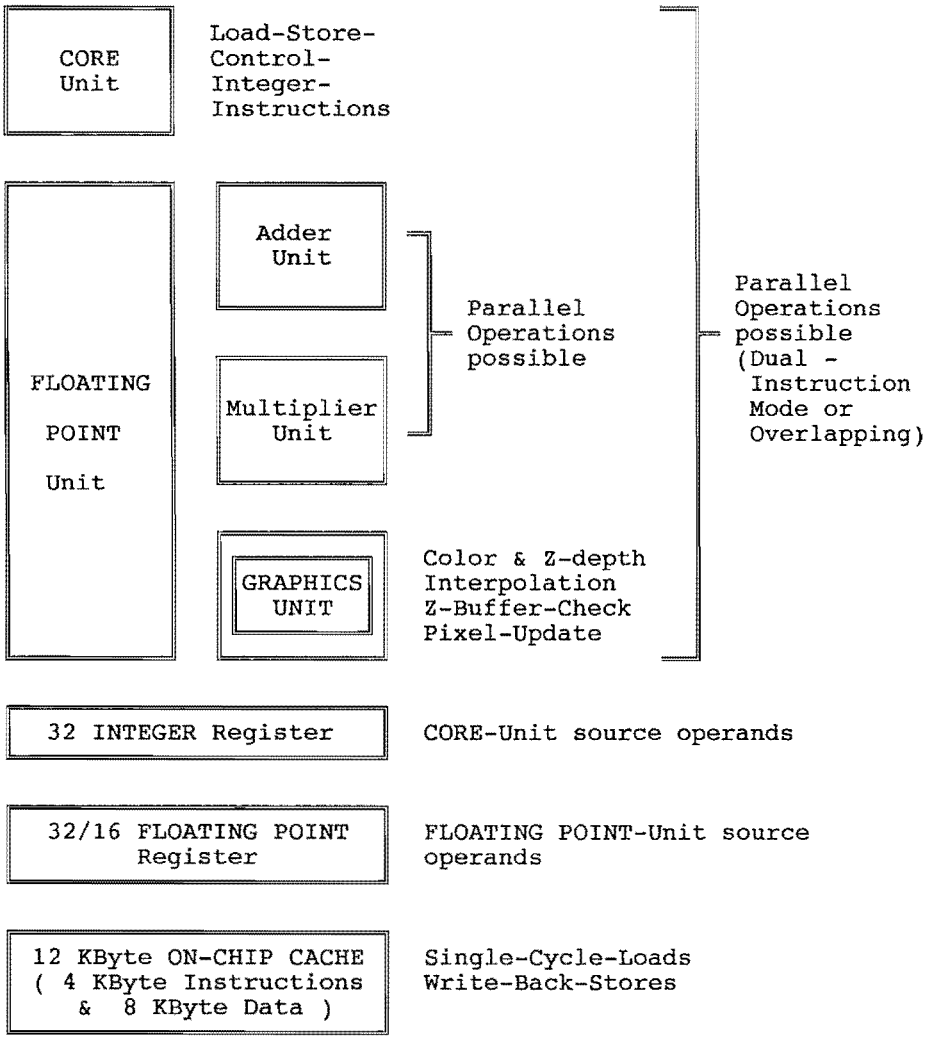

Figure 1: Hardware Layout of the INTEL I80860

Operating the I80860 in dual instruction mode, where a core unit instruction (integer arithmetic or load–store instructions) can be executed parallel to a floating point instruction (FP–Add or FP–Multiply or GU–instruction) is not supported by the standard compilers. This is not always a big disadvantage because the I80860 handles overlapping instructions itself, e.g. a floating point instruction which takes 3 cycles for completion maybe followed by 2 core instructions without additional time penalties, i.e. all three instructions will have been executed after the 3 clocks. For the assembly programmer it is nevertheless save and easy to use the dual instruction mode if he desires, especially if the pipelined versions of the floating point instructions are used. These are effectively single clock instructions and no overlap–effect can occur. The only way to achieve parallelism between core and floating point instructions is then the dual instruction mode.

When used as a vector processor, the I80860 needs (as all vector processors) special preparation of data, like gathering a large amount of input data before processing them in a program using pipelined instructions, or scattering these data after the processing etc.. With its comparatively small data cache (8KByte) and its medium sized register set (32 integer and 30 single or 15 double precision floating point registers) the I80860 is by design not a load–store machine. The investigations were performed on a KONTRON SCB860 AT–Add–On board. When this investigations were made, the on–board memory could not be operated at full speed for several reasons (only A2–steppings of the processor were available, no NENE = NExtNEar signal was generated, not the fastest RAMs were available etc.). So the data preparation turned out to be quite costly (in terms of execution speed) in some cases and the time used for this data preparation was included in the following execution speed investigations. One should keep that in mind before discarding vectorised algorithms which may not be as satisfactorily as expected (or hoped). Additionally it is not guaranteed that the presented algorithms are the fastest possibilities, especially if they involve a lot of different memory accesses. Sometimes regrouping of memory accesses is of major impact on the execution time and "programming by the book" (e.g. the "Programmer's Reference Manual"[1]) is less important than minimisation of page faults etc.. Nevertheless I have not always bothered to try every combination as long as the performance was not too unsatisfactorily.

2. Description of the Basic Algorithm

From the nature of the GU instructions as described below, it becomes clear that the supported algorithms are scanline–based algorithms for generating smooth–shaded (Gouraud–shaded) polygons. When confronted with such a powerful floating point machine one might wish to use more sophisticated rendering algorithms, like ray–tracing, radiosity algorithms etc.. It should not be excluded, that the GU may be useful for such algorithms as well, but this is far from obvious. The aim of this

article is to investigate the capabilities of the GU and I will therefore restrict myself to the algorithm for which the GU instructions are obviously tailor–made.

The examinations only consider the basic interpolation, Z–Buffer and pixel–update mechanism which is typical for pixel oriented shading algorithms (e.g. scanline–algorithms). For a certain amount of pixels, the color and z–depth values are calculated by linear interpolation. After that the z–depth value for each pixel is compared with the corresponding Z–Buffer value and if necessary, the color and Z–Buffer values for this pixel are updated. The algorithm will be referred to as painting or rendering algorithm.

3. GU Instructions and Their Data Formats

The GU of the I80860 supports several different data formats. In general the color and z–depth values are "floating–point" like in the sense that they consist of an integer and a fractional part "iiii.ffff" (fixed–point format). Depending on the size one or more of the values are stored in a 64–Bit word. After the interpolation they are truncated to their integer part and packed into a special "MERGE" register from where they can be transferred to an ordinary FP–register and then stored or compared to similar values (Z–Buffer check). The supported formats are:

("fs1", "fs2" and "freg" stand for any double precision FP–register and "(p)" indicates the modified instruction for the pipelined version, if available)

3.1 8–Bit Pixel

The interpolation values are 16–Bit deep with an 8–Bit integer part and an 8–Bit fractional part. The interpolation instructions are:

"(p)faddp fs1,fs2,freg".

3.2 16–Bit Pixel

The interpolation values are 16–Bit deep with a 6–Bit integer and a 10–Bit fractional part. A 16–Bit pixel is understood to be a RGB–value with a 6–Bit R–part, a 6–Bit G–part and a 4–Bit B–part packed into a 16–Bit word. The instructions are the same as for the 8–Bit pixel, but will behave differently during the merging process. (The instructions consult a bit field in one of the control registers, where the pixel size has to be specified.)

3.3 32-Bit Pixel

Again the pixel value is an RGB-value, where each part covers 8 bit and the most significant byte is void. The interpolation value for each color is 32-Bit deep with a 8-Bit integer portion and a 24-Bit fractional part. Again the instruction is the same and the merging will be performed appropriately.

3.4 16-Bit Z-Buffer

The instruction

"(p)faddz fsl,fs2,freg"

performs a 32-Bit z-depth interpolation, where the results are truncated to their 16-Bit integer part. The

"(p)fzcks fsl,fs2,freg"

instruction performs the corresponding Z-Buffer check. This instruction stores a 64-Bit word which contains the four smaller z-values. Additionally it sets a Pixel-mask bit field in a control register which tells a consecutive pixel store instruction

"pst.d fsl,target-address"

which pixels should be updated and which not.

3.5 32-Bit Z-Buffer

If 32-Bit interpolation is required, the usual integer adds can be used. For 64-Bit interpolation (32-Bit integer and 32-Bit fraction) the

"(p)fiadd.dd fsl,fs2,freg"

instruction is available. The corresponding Z-Buffer check instruction is

"(p)fzckl fsl,fs2,freg".

3.6 Possible Combinations

The following combinations are possible (in principle, but not all are reasonable):

I) 8-Bit PIXEL & 16-Bit Z-BUFFER.

II) 8-Bit PIXEL & 32-Bit Z-BUFFER (32-Bit interpolation)

III) 8-Bit PIXEL & 32-Bit Z-BUFFER (64-Bit interpolation)

IV) 16-Bit PIXEL & 16-Bit Z-BUFFER.

V) 16-Bit PIXEL & 32-Bit Z-BUFFER (32-Bit interpolation)

VI) 16-Bit PIXEL & 32-Bit Z-BUFFER (64-Bit interpolation)

VII) 32-Bit PIXEL & 16-Bit Z-BUFFER.

VIII) 32–Bit PIXEL & 32–Bit Z–BUFFER (32–Bit interpolation)
IX) 32–Bit PIXEL & 32–Bit Z–BUFFER (64–Bit interpolation)

4. Realisations of the Basic Algorithms

Three different types of programs will be analysed and examined in the following context:

a) Pure C–Versions (PCV)

b) C–Frames with hand–optimised Core–routines (OCV)

c) C–Frames with hand–optimised Graphics–Unit–routines (OGV)

Before the presentation of the results, some general remarks on the level of optimisation and the complexity of the programs are necessary.

The accuracy of the a) and b) versions is much higher, because all interpolations are done in double precision floating point format. It is questionable whether this is really an advantage or only an assurance, but it is mentioned for the sake of honesty.

4.1 PCV

The C–routines are in general very simple and leave not much space for optimisation or variation. The only decision one has to make is whether to do the color interpolation always or only if the Z–buffer check yields a positive result.

E.g. USE :

```
for(i=0;i<no;i++) {
  z += z_delta;
  col += col_delta;
   /* Color interpolation */
  if( (zs = (short)z) < zbuf ) {
    zbuf[i] = zs;
    pixel[i] = (char) col;
  }
}
```

OR USE:

```
for(i=0;i<no;i++) {
  z += z_delta;
  if( (zs = (short)z) < zbuf ) {
    m = (double) i;
    zbuf[i] = zs;
    pixel[i] = (char)(col_start + m*col_delta);
  }
}
```

The first version will always do one floating–point add, whereas the second version will do instead only an occasional integer–to–double conversion in

connection with an floating-point add and multiply. For the 8-Bit-per-Pixel cases both versions are equally fast. If more than one color value has to be interpolated, as for the RGB-cases (16- and 32-Bit-per-Pixel), the second version becomes more effective and was used in the comparisons.

4.2 OCV

In this versions the C-frame program is the same as for the PCVs, only the core-program, e.g. the "for"-loop described above, is substituted by a hand-optimised assembly routine. These routines are not too complex and any programmer with a bit of experience will not encounter greater difficulties. Again the routine for the RGB-cases is more complicated, because in this cases it pays to use pipelined floating point instructions and consequently dual instruction mode as well.

4.3 OGV

In order to use the graphics unit effectively, a lot of more or less complicated preparations are necessary. For all cases the pixels have to be treated in 8-Pixel packages to ensure that the Z-buffer check instructions and the pixel stores are synchronous.

4.3.1 Preparation of the Starting Values

All starting values have to be prepared and converted into the data formats described above. This is trivial and not time critical because it applies only for the starting and delta values.

Example:

```
/* 8-Bit-Pixel color starting value   */
/* assuming all 8 pixel use the same delta
increment */
/* "shift" double value by 8 Bit and truncate
to short */
col = (short) (col_start * 256.0);
cold = (short) (col_delta * 256.0);
```

```
/* Accumulate 8 pixels in two doubles */
double col_array[2];
cpt = (short *) col_array;
/* first the even pixels */
cpt[0] = col - 8*cold;
cpt[1] = col - 6*cold;
cpt[2] = col - 4*cold;
cpt[3] = col - 2*cold;
/* then the odd pixels */
cpt[4] = col - 7*cold;
cpt[5] = col - 5*cold;
cpt[6] = col - 4*cold;
cpt[7] = col - 1*cold;
/* Accumulate the corresponding delta values */
double cold_array;
cdpt = (short *)&cold_array;
/* Even and odd pixels use the same delta */
cdpt[0] = 8*cold;
cdpt[1] = 8*cold;
cdpt[2] = 8*cold;
cdpt[3] = 8*cold;
```

Similar preparations have to be done for the z-depth interpolation or in cases with different pixel depths.

4.3.2 Preparation of the Delta Switches

Under realistic assumptions it is not likely that the interpolation uses a constant delta value for more than 10 pixels (typically the distance from one edge to the next). The optimised routines are only efficient if much more pixels can be treated successively, at least a few 8-pixel blocks are required. Interpolating the pixels across a whole object surface would meet this requirement more likely, but to guarantee the correct interpolation it must be possible to switch from one delta value to the next. During my investigations this problem turned out to be very crucial because without a very efficient solution to this problem, the most efficient OGV routine might be completely worthless and even slower than the PCV. The next chapter will give a more detailed description of this problem and offer some solutions as well.

In general the data preparation and the algorithm depend on the chosen model (cases I) – IX)) but the overall structure is always the same :

1st Step:

Do all interpolations for color and z-depth pipelined if possible and do all required loads and stores in parallel. The special interpolation instructions merge the results as described in [1]. For 8-Bit pixels one has to interpolate alternating between even and odd pixel-numbers, always 4 Pixels per instruction. For 16-Bit and 32-Bit pixels, one interpolates 4 successive pixels per instruction, but alternating between R-, G- and B-interpolation.

2nd Step:

Do the required Z–Buffer checks for all 8 Pixels to prepare the Pixel–Mask for the subsequent stores.

3rd Step:

Do the pixel stores which interpret and reset the Pixel–Mask. This step may be executed parallel to Step 1, for overlapping loops. In this case the stores would complete the calculations of the PREVIOUS iteration and must be performed before the new Z–Buffer checks affect the Pixel–Mask setting. For steps 2 and 3 see figure 2.

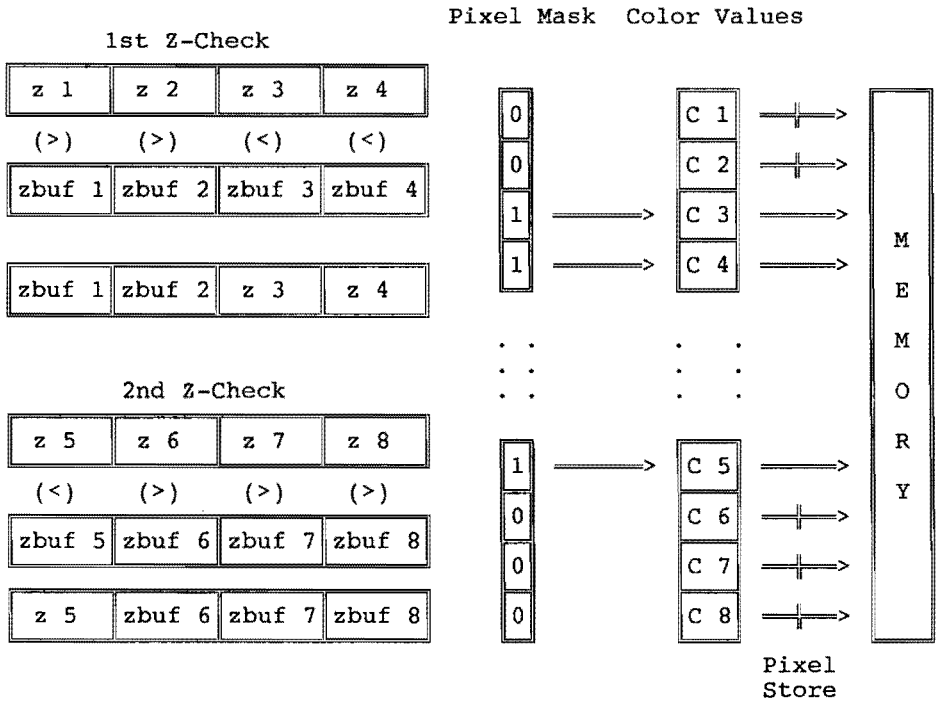

Figure 2: Z-Buffer-Check and Pixel-Store

5. The Delta–Switch Algorithm

The basic assumption for all the following considerations is that the calling program supplies all different delta values which occur along the intersection of the scanline with the object surface. This is a reasonable assumption and means no additional effort compared to the simple C–program versions, even if the data preparation is a bit more complex.

With every iteration, the OGV–Program will now load a certain delta value, use it for interpolation and then load the next value. This may be the same as the first one or a different new delta value, if at this point a delta switch is required. To avoid duplication of data, one could realise this behaviour by gathering the data into a delta structure (this should happen during the preparational period) and supplying a pointer array to the OGV–routine.

In this array a pointer to a certain delta structure appears as often as this structure is needed for interpolation. A delta switch is then nothing else than changing from one pointer to the next. In reality this procedure is not easily implemented for the following reason: All delta–values have to be gathered in double (64–Bit) words, because all pixel manipulation instructions operate on double words.

In detail this means:

Four 16–Bit Z–depth or color values, or two 32–Bit Z–depth or color values, or eight 8–Bit color values must be accumulated in a double word. Depending on the combination of Z–Buffer and color values, such a delta structure would contain up to 8 different delta values. Taken into account that one delta value might be used only for about 10 Pixels, it becomes clear, that in this case the preparation of the delta structures does not lead to data multiplication. Consequently, for each different model one has to find the best suited solution, which in general is a mixture of using extended delta–value arrays and delta–structure–pointer arrays. (The extended delta–value array maybe a structure array as well, so the emphasis is put on the term "pointer" in the second expression.)

Timing these models may be difficult, because it is not always obvious, which preparations are really additional efforts and must be included in the execution time measurements. So for the three examples which are considered below, the algorithm which is used is explained in detail. Further clarification is given which parts of the algorithm are included in the execution time measurements. It is common to all applications using "vectorised" special purpose routines, that the data preparations should be subject to careful considerations, because otherwise they might cost more than the optimised routines gain.

5.1 Examples

With 16–Bit pixels and 16–Bit z–Buffer the smallest complete delta structure would consist of:

```
2 double words, each containing two z-deltas
3 double words, 1 for each color with 4 deltas per double word
-------------------------------------------------------------
10 words per structure.
```

If the average pixel chain length is 200 pixels with a delta switch every 10th pixel, this would require a total of 30 different delta structures. For two successive sets of pixels one would need two "pure" structures which will be used twice and one "transition" structure containing the delta switch, i.e. a total of 20 pure structures and 10 transition structures. Compared to the original set of different delta values (20 x 2.5 words) the new set requires 6 times the memory space and the corresponding load–store operations.

The pointer–array method would need this set of different structures and additionally a pointer array, where each pure structure is referenced twice and each transition structure only once. This would mean another 50 words of memory space and the corresponding store operations. An asynchronous treatment of z–delta and color–delta values would be another possibility one could think of, but for the model with 32–bit z–interpolation this is disadvantageous :

```
30 different color-delta structures, 6 words each = 180 words
50 color-delta-pointers,             1 word each  =  50 words
20 different z-delta values,         2 words each =  40 words
100 z-delta-pointers,                1 word each  = 100 words
-------------------------------------------------------------
TOTAL                                             = 370 words
```

compared to 350 words for the simple pointer array.

For the extended delta–value array method the pure structures would have to be duplicated to yield the complete set of 50 delta structures. The additional effort would amount to 200 words memory space accompanied by the corresponding load–store operations. In this example the pointer method should prove to be superior, as it is probably for all cases, where the pixel size is larger than 8 Bit. Only if the pixel subsets along which a delta value remains constant become very small (e.g. 4 pixels or less) the delta–value array method will be advantageous. In this case the set of different delta structures will already be the extended delta–value array.

5.2 Summary

To enable efficient usage of optimised pipelined pixel operations it is necessary to maintain a continuous input flow of delta values for the interpolations. If the usage of extended delta–value arrays does not involve to many preparational load–store operations it is a problem independent solution which is easy to include in the execution time measurement.

The usage of pointer–arrays will involve some additional load–store operations as well, but this effort is difficult to estimate. If the number of delta switches is

large the method is not very efficient, especially if the values for more than 2 pixels must be packed into double words.

6. Results

Three of the 9 different cases mentioned above (cf. 3.6) have been investigated for the following reasons:

a) Case I:

This case is the classical example for using the delta–value method of implementing the delta–switch algorithm.

b) Case IX:

This case is quite the opposite to Case I) and is best suited for the pointer–array approach. The time measurements are based on the simplified model described in chapter 5.1 (Example).

c) Case V:

Because of the simple z–interpolation this case yields the fastest OCV programs and offers therefore the biggest challenge for the Graphics Unit.

For the OCV and the PCV programs the execution time depends on the size of the screen and the number of substitutions due to positive Z–Buffer compares. The latter is also a function of the screen size, if the screen size is small, the Z–Buffer becomes loaded with small values quite soon and the number of substitutions decreases. The screen size for PCV and OCV programs was 768 x 1024 pixels. The execution time of the OGV is independent of the number of substitutions, because the store instructions will always be executed, but realised only where the Pixel–Mask–Bits are set.

The execution time measurements presented below, are given for various pixel chain lengths, from 10 to 1000. Only for the OGV programs pixel chain lengths larger than 10 may model real situations. For OCV and PCV programs the corresponding figures are given only for the sake of completeness.

Each run computes 100,000 different pixel chains of the given length, with the Z–depth values randomly distributed. To enable a correct comparison it must be noted that e.g., 100,000 repetions of pixel chain length 200 for the OGV are equivalent to 2,000,000 repetitions of pixel chain length 10 for the PCV and OCV.

The EQUIVALENCE COMPARISON after each table gives the corresponding figures. The number of substitutions for the OCV and PCV programs is app. 2,200,000 in this case.

6.1 Case I)

No of repetitions : 100,000

Table 1

No of Pixels	OGV	OCV	PCV
10	----	940	1820
20	1260	1600	3130
40	1430	2850	5490
100	3900	6430	11860
200	6540	12350	21810
1000	40100	58710	99420

All timings are given in Milli-seconds.

EQUIVALENCE COMPARISON:

OGV : 100,000 x 200 in 6540 msec

is equivalent to

OCV : 2,000,000 x 10 in 15680 msec

PCV : 2,000,000 x 10 in 25350 msec

with 2,200,000 substitutions.

In terms of Gouraud-shaded polygons, the latter results may be translated into:

OGV : 30,500

OCV : 12,750

PCV : 7,900

100-Pixel GOURAUD-SHADED POLYGONS per SECOND.

6.1.1 Remarks

The delta-switch algorithm is based solely on delta-value arrays, because in this case each delta-structure would contain the delta values for 8 successive pixels gathered in a total of 4 double words.

Optimised copy routines are used to distribute and repeat the delta values appropriately. The time needed for this copies is INCLUDED in the execution time measurements. Not included is the preparation of the index array, which contains the number of pixels for which each delta value is valid. This array is used by the optimised copy routines. (The array contains nothing else than the number of pixels

between an edge pair and is therefore assumed to be supplied by the calling program, which will store these data anyway.)

The original values are assumed to be supplied in two different arrays, one containing the 32–Bit z–delta values (format: 16–Bit integer and 16–Bit fractional part) the other containing the 16–Bit color values (format: 8–Bit integer and 8–Bit fractional part). Two different copy routines are required to generate the extended arrays.

The OCV program uses only simple optimisation, because of the small amount of floating point calculations. Pipelining and dual instruction mode are no improvement in this case and the optimisation is achieved by ordering and minimising the number of instructions.

6.2 Case IX)

No of repetitions : 100,000

Table 2

No of Pixels	OGV	OCV	PCV
10	-----	990	2810
20	1320	1590	4610
40	1760	2800	7470
100	4220	6150	13790
200	8070	11480	22680
1000	61130	53160	86340

All timings are given in Milli–seconds.

EQUIVALENCE COMPARISON:

OGV : 100,000 x 200 in 8070 msec

is equivalent to

OCV : 2,000,000 x 10 in 15640 msec

PCV : 2,000,000 x 10 in 25260 msec

with 2,200,000 substitutions.

In terms of Gouraud–shaded polygons, the latter results may be translated into:

OGV : 24,800

OCV : 12,750

PCV : 7,900

100–Pixel GOURAUD–SHADED POLYGONS per SECOND.

6.2.1 Remarks

In this case all delta values are gathered into one large delta structure of the type:
"struct delta { double zl–delta, z2–delta, R–delta, G–delta, B–delta };"

This delta–structure array is twice as large as the original, because two color values have to be accumulated in one double word. It is clear that this method simplifies addressing, because all delta values are close together and some time will be saved, due to cached loads in the OGV–Program.

The execution time measurement INCLUDES the time needed to produce the delta structures from the original data.

The example of chapter 5.1 is of course only a very crude model for reality. One of its artificial properties is that it needs no transition delta structures, but even in real situations there should be not too many of these structures. Provided the program permits an easy way to insert such transition structures (e.g., by adding a pointer "struct delta *next" to the "struct delta" so that the different structures are forming a chain rather than an array), this means not much additional effort. One could use the optimised routine to produce the pure delta structures by duplication and afterwards insert the transition deltas at the appropriate locations.

Additional investigations have shown that an asynchronous treatment of the z–delta and color–delta values is disadvantageous, it needs more memory space and is slower.

The PCV program is slower as in case I) as one could expect because of the additional effort of interpolating three color values. In case of the EQUIVALENCE COMPARISON this difference becomes neglectable due to the large number of memory accesses.

The OCV program is as fast as in case I), because the larger amount of floating point operations allows the efficient usage of pipelined instructions in connection with dual instruction mode.

The OCV becomes than more or less independent of the number of substitutions, because the almost all calculation are finished anyway before the result of the Z–Buffer check affects the color interpolation. (The Z–Depth value has to pass through the Adder–Pipeline twice, before the Z–Buffer check can be performed.)

6.3 Case V)

No of repetitions : 100,000

Table 3

No of Pixels	OGV	OCV	PCV
10	----	880	2580
20	1810	1380	4230
40	1490	2250	6420
100	4120	4510	10880
200	5990	7960	16360
1000	63770	34270	52220

All timings are given in Milli–seconds.

EQUIVALENCE COMPARISON:

OGV : 100,000 x 200 in 5990 msec

is equivalent to

OCV : 2,000,000 x 10 in 11860 msec

PCV : 2,000,000 x 10 in 18200 msec

with 2,200,000 substitutions.

In terms of Gouraud–shaded polygons, the latter results may be translated into:

OGV : 33,300

OCV : 16,850

PCV : 11,000

100–Pixel GOURAUD–SHADED POLYGONS per SECOND.

6.3.1 Remarks

With 2 z–delta values or 4 color values per double word the complexity of this model lies between the previous cases.

The 32–Bit Z–Buffer with 32–Bit interpolation means that simple integer adds can be used for the z–depth interpolation. This has its effect mainly on the OCV–program, were z–interpolation can now be performed parallel to the color interpolation and no float–to–integer conversion for the z–values is required. This explains the extraordinarily good OCV performance. More or less the same goes for the PCV program.

The OGV program behaves a bit irregular. The drops in execution time for pixel chain length 40 and 200 are due to the fact that these numbers are multiples of 8. This means that all pixels are updated with the optimised algorithm. If the pixel chain length is less than 16 and no multiple of 8, all or at least the modulo–8–rest pixels are treated as in the PCV program. The amount of optimisation for the OGV program must be very large to achieve a better performance as the OCV. Again the method of the delta–structure–pointer array proved to be most efficient.

I have tried several other possibilities as well, e.g. separate z–depth interpolation, delta–value arrays etc., but the pointer array method gave the best result, if a high level of optimisation is provided. Not only the pixel update routine has to be optimised, but the data preparation routines as well.

The execution time measurement INCLUDES the complete preparation of the delta structures, which in this case means transition structures as well, and the generation of the pointer array. The underlying model is exactly the model described in chapter 5.1, which means that it is still a simplification. For a real model the optimisation of the data preparation routines becomes a bit more complex, but the strategy may be the same:

1. step :

Generate all pure delta structures. This is simple and easy to optimise. The delta structures should be connected via pointers and may include the number of occurrences (i.e. how often this structure is to be used for interpolation).

2. step :

Insert all transition structures. For the simple model this is schematic as well and therefore easy to optimise. For real situations it may become tedious, but nonetheless the number of transition structures should remain small. Therefore it consumes not to much execution time, even if it is programmed in C–code.

3. step :

Generate the pointer array to be used by the pixel update routine. This can be optimised using different strategies, each based on the occurrency information contained in the delta structures.

The OCV and PCV routines profit from the very simple Z–Buffer interpolation. Especially for the OCV routine which is in principle the same as for case IX) the benefit is large. The z–depth interpolation is now a Core operation and can be performed parallel to a floating point operation. The Z–Buffer check can therefore be performed much earlier and consequently for negative compares aborts the color interpolation much earlier.

7. Miscellaneous

The rendering algorithm is only one example, where the Graphics Unit instructions are useful. One may think of other examples, where the interpolation features are useful as well, e.g. in line drawing algorithms.

But one should keep always in mind, that the conversion of the input data (which may be in "float" or "double") to the special format needed for the interpolation instructions requires additional effort. It is therefore only efficient if the subsequent calculations make extensive use of the converted data.

In general preparation of a single data item requires

1 floating point load of the input values and

1 floating point multiply,

in order to "shift" the input values and save the fractional part after

1 truncation to an integer format

and 1 integer store is performed.

(Almost always some additional manipulations are necessary, because the interpolation instructions of the GU do more than one interpolation with each instruction. This means one has use larger increments, more than one starting value, etc..)

Example:

16 Bit Z-Buffer value:

```
double input;
long output;
/* Shift by 16 Bit before conversion */
output = (long) (input * 65536.0);
```

In optimised code the conversion takes between 4 and 7 clock cycles, depending whether the load and store hit the data cache or not. For the application this means, that any program not using the GU may take 4 to 7 clock cycles more time. If one uses highly optimised assembly code this could be equivalent to 4 to 7 operations in dual instruction mode. For a simple line drawing algorithm, i.e. the interpolation between a starting point and an end point on the screen, a optimised routine using the GU is not necessarily faster than an optimised routine not using the GU, but the latter requires less programming effort.

In this case the scale could tip towards the GU program, if the lines to be drawn are quite long (e.g. in the range of 100 Pixels) or if additional use of the data would involve compares and corresponding pixel updates, etc.. One might think of a clever way to implement line styles this way or something like that.

This issue will not be pursued any farther in this paper, it is just an example for the general rule:

The usage of the Graphics Unit's interpolation and merging instructions is only reasonable, if the calculation effort justifies the data conversion.

8. Conclusions

With a considerable amount of optimisation effort it is always possible to accelerate the painting algorithm using the GU instructions in a special assembly routine.

In general this goes hand in hand with optimised data preparation routines and results in complex algorithms. The acceleration factors lie between 2 and 3 for OCV and between 3 and 4 for PCV routines.

The execution time does not depend much on the complexity of the underlying models, for the examined cases it was more or less in the same range. E. g. the choice between 8–Bit pixels and 16–Bit or 32–Bit pixels affects the complexity but not the performance of the resulting program.

The OCV method offers an efficient and comparatively simple possibility for optimisation, if the problem is not too time critical or if development time is more critical than performance. Especially for the simple Z–Buffer interpolation it is a good alternative.

When operated as a graphics engine, the GU of the Intel I80860 is only of limited use, ie. for 2D– and 3D–Output primitives of the kind described above. But it may play a decisive role if the principle "Simple Things should be Fast" is to be fulfilled, even so it requires a lot of optimisation.

9. References

1. i860 64-Bit Microprocessor Programmer's Reference Manual. Intel Literature Sales, Santa Clara, CA 95052-8130,1989