SSE2 Optimization – OpenGL Data Stream Case Study

By

Arun Kumar
Intel Corporation

Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

TABLE OF CONTENTS

Abstract

At its core, Streaming Single Instruction Multiple Data Extensions (SSE2) aims to encourage exploitation of parallelism. The SSE2 benefit is allowing an application to perform the same manipulations on more than one data item at a time. To take advantage of SSE2 the software developer should be on the lookout for situations where computations can be done in parallel on multiple data items. This paper explores the use of SSE2 for a specific case in which bounding boxes are computed for each triangle in an input graphics data stream. First, the case of a stream of disjoint triangles is considered and two different ways of approaching an SSE2 implementation are demonstrated, highlighting the benefit of one over the other. Next, the case of triangle strips is examined and an SSE2 implementation is developed. Performance of the approaches developed is compared.

OpenGL Data Stream

The data being considered in this paper is an OpenGL stream of triangle data. Triangles are used in a variety of ways by graphics applications. For example, they could be part of a surface tessellation (a specific case of a more general polygon representation of a surface), or they could be the representation of a volume (each triangle bound by a unit volume). Depending on an application's requirement, vertices of the triangle typically may have additional data associated with them (i.e. color coordinates, texture coordinates, lighting coordinates). This paper examines the task of computing the smallest box that bounds a given triangle. The size of the smallest bounding box along with coordinate positions of the box allows the study of a number of interesting properties of 3D models, for example, intersecting surfaces, distance of closest approach, intersection of a ray with a surface etc. This paper demonstrates two ways of implementing SSE2 optimizations to this problem and shows that although both are good optimization approaches, one out-performs the other.

Bounding box computation

Consider a triangle (A,B,C)=((ax,ay,az),(bx,by,bz),(cx,cy,cz)) as shown in Figure 1. To find out the smallest possible box that contains the triangle, we need to find the principle lengths of the box. Thus, we need to perform the following calculations:

Box_size_x = Abs(Xmax-Xmin)

Box_size_y = Abs(Ymax-Ymin)

Box_size_z = Abs(Zmax-Zmin)

Where,

Xmax = Max(ax,bx,cx) and Xmin = Min(ax,bx,cx)

Ymax = Max(ay,by,cy) and Ymin = Min(ay,by,cy)

Zmax = Max(az,bz,cz) and Zmin = Min(az,bz,cz).

Abs() is the usual absolute value function, while Min() and Max() are functions that compute the arithmetic maximum and minimum of the set of values passed to them. Note that the operations performed in the x-coordinate direction are independent of those in the y-coordinate and z-coordinate directions. Thus parallelism could be used to speed up the entire bounding box computation.

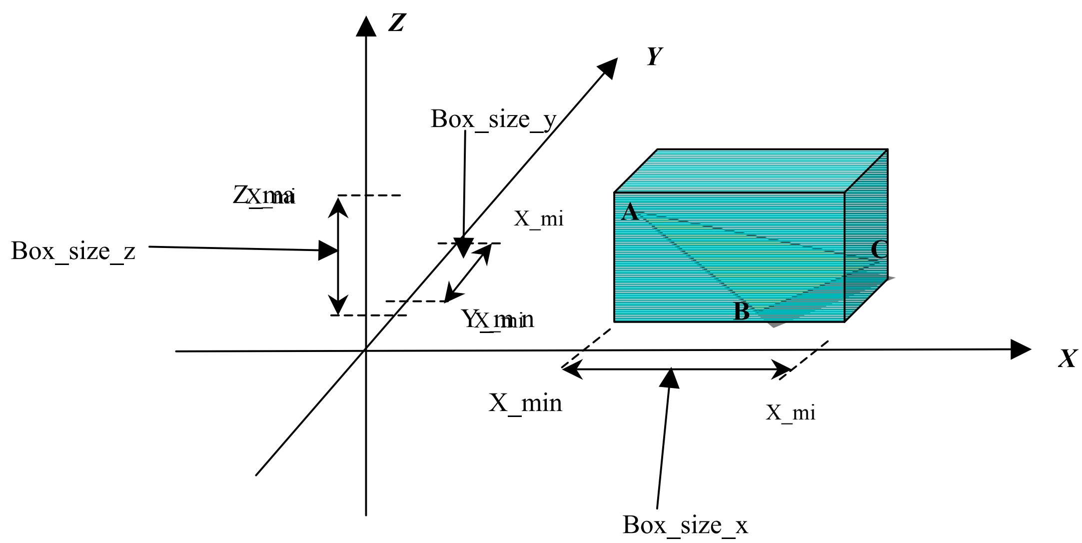

Figure 1 Bounding Box for a Triangle – Coordinate geometry

Using SSE2, the simplest approach

This section describes the first and possibly the simplest approach in implementing SSE2 optimizations. The input data stream has the following form: {Triangle1, Triangle2,Triangle3…}~{(ax1,ay1,az1,xx,xx,xx),(bx1,by1,bz1,xx,xx,xx),(cx1,cy1,cz1,xx,xx,xx),(ax2,ay2,az2,xx,xx,xx),(bx2,by2,bz2,xx,xx,xx),(cx2,cy2,cz2,xx,xx,xx),(ax3,ay3,az3,xx,xx,xx),(bx3,by3,bz3,xx,xx,xx), (cx3,cy3,cz3,xx,xx,xx)….}. Note that all vertices have associated additional data (indicated by "xx"), which is unimportant for the purposes of this calculation. The stride is defined as the distance in bytes between the start of one vertex and the start of the vertex immediately succeeding it in the stream. Assuming floats in this case the stride would be 24. Loading 3 vertices for a triangle with three SSE2 reads is depicted in Figure 2, since it is possible to read four floats worth of data with one SSE2 load instruction.

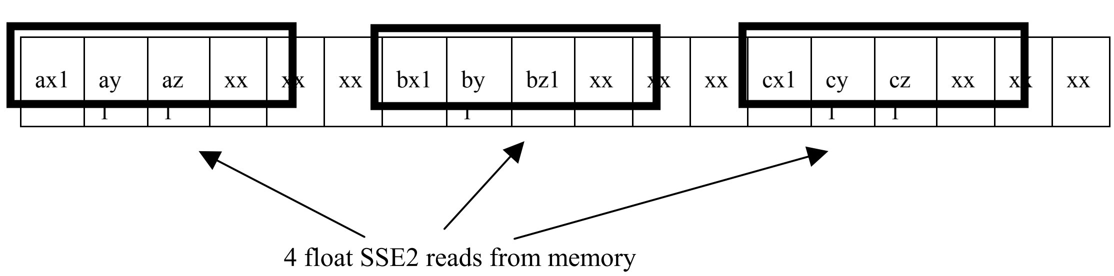

Figure 2 OpenGL triangle stream and SSE2 loads

The SSE registers now contain:

```
xmm0 = {ax1, ay1, az1, xx}

xmm1 = {bx1, by1, bz1, xx}

xmm2 = {cx1, cy1, cz1, xx}
```

We can perform our planned operations to find the bounding box as follows:

```
xmm3 = max(xmm0,xmm1)

xmm4 = max(xmm3,xmm2)
```

and,

```
xmm5 = min(xmm0,xmm1)

xmm6 = min(xmm5,xmm2)
```

functionally, the end result is:

```
xmm4 = Max(x,y,z) // max_vec

xmm6 = Min(x,y,z) // min_vec
```

The bounding box can now be finalized by rearranging data so that before it is written out to memory we have the data arranged in the following format as shown in Figure 3: (min_vec.x, min_vec.y, min_vec.z, max_vec.x, max_vec.y, max_vec.z).

Typically, applications require the bounding box data in a normalized short integer (or perhaps byte) format. Before the final coordinates are written out to memory, the floats must be converted to suitable integers and then appropriately clamped within a certain range (e.g. the range (min,max) where min and max are usually integer values with application dictated precision). One way to obtain appropriate clamping values in SSE2 registers is to first declare aligned arrays with the clamp values as elements:

```
int declspec(align(16)) clamp-min = {min,min,min,min};

int declspec(align(16)) clamp-max = {max,max,max,max};
```

Then load these arrays into XMM registers using movaps. Assume that `clamp-min` and `clamp-max` arrays are loaded into xmm6 and xmm7.

The process of clamping is simply carrying out the following operations:

```
xmm8 = Max(xmm6,xmm4)

xmm8 = min(xmm8,xmm7)  // for the max_vector

xmm9 = Max(xmm6,xmm5)

xmm9 = min(xmm9,xmm7)  // for the min_vector
```

Since there are only 8 XMM registers on the Pentium® 4 processor, the resources required in the pseudo-code above exceed the resources at our disposal. This means that registers no longer needed will have to be reused.

It is quite common for applications to store the bounding box output data in fewer bits of precision than the usual 32 bit integers (this of course depends on how the application intends to use the integer data). Based on a real world application, 10 bits of precision for each component is assumed in this paper partly because it makes the process of packing the data more instructive. With this assumption, the clamping max value is 1023 (1111111111, binary). While the conversion to integers and clamping can be performed in SSE2 registers it would be instructive to draw attention to the fact that the MMX™ registers can also be used and in fact can help reduce register contention among the SSE2 register set. Figure 3 details how the MMX registers can be loaded with the appropriate data two dwords at a time.

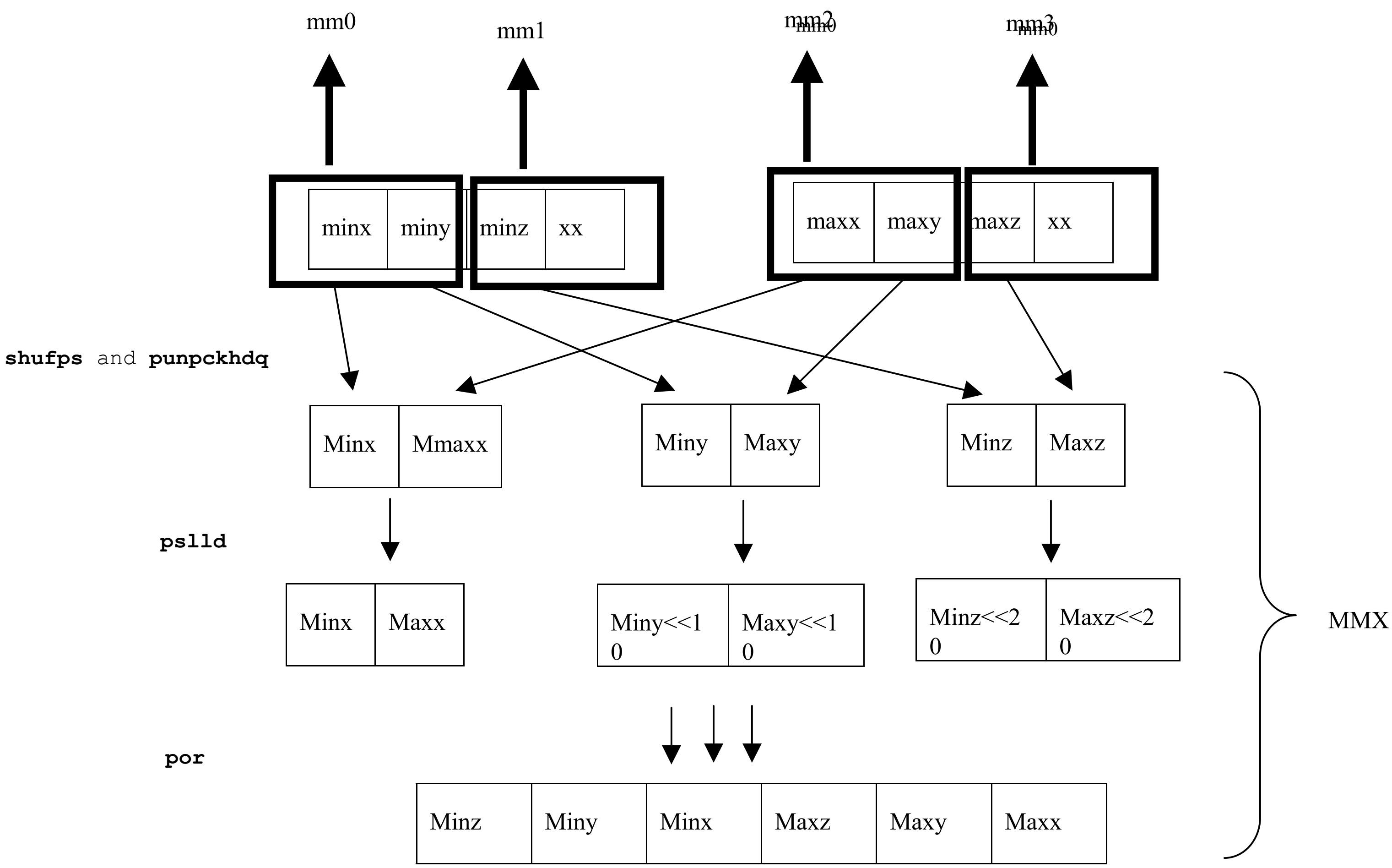

Figure 3 MMX for assembling Bounding boxes

First the data corresponding to x,y,z coordinates is shuffled and packed into separate MMX registers. The next operation shifts the data appropriately (y coordinate data shifted left by 10 bit position while z coordinate data shifted by 20). Finally the 3 MMX registers are ORed. The data is now ready to be written to memory.

Inefficiency in this approach

The approach just described will provide a significant performance boost. However there is some inefficiency in this approach since the SSE2 mathematical operations are working on 3 out of 4 slots in each XMM register. One slot remains filled with unimportant data. In fact, care has to be exercised about the kind of data in the unused slot since denormal data could incur SSE assists slowing down the calculations. If the fourth unused slot can be utilized in each of the operations, it could boost the output of the computations we are performing. Processing data containing four triangles at a time allows us to completely utilize the SSE2 registers. This approach is discussed in the next section.

Using an improved SSE2 approach

An improved SSE2 approach processes four triangles at a time. In this section, it is shown that this ensures the unused slot in the XMM registers is used. Compared to the approach of the previous section it gives a performance improvement. Figure 4 shows how this method works.

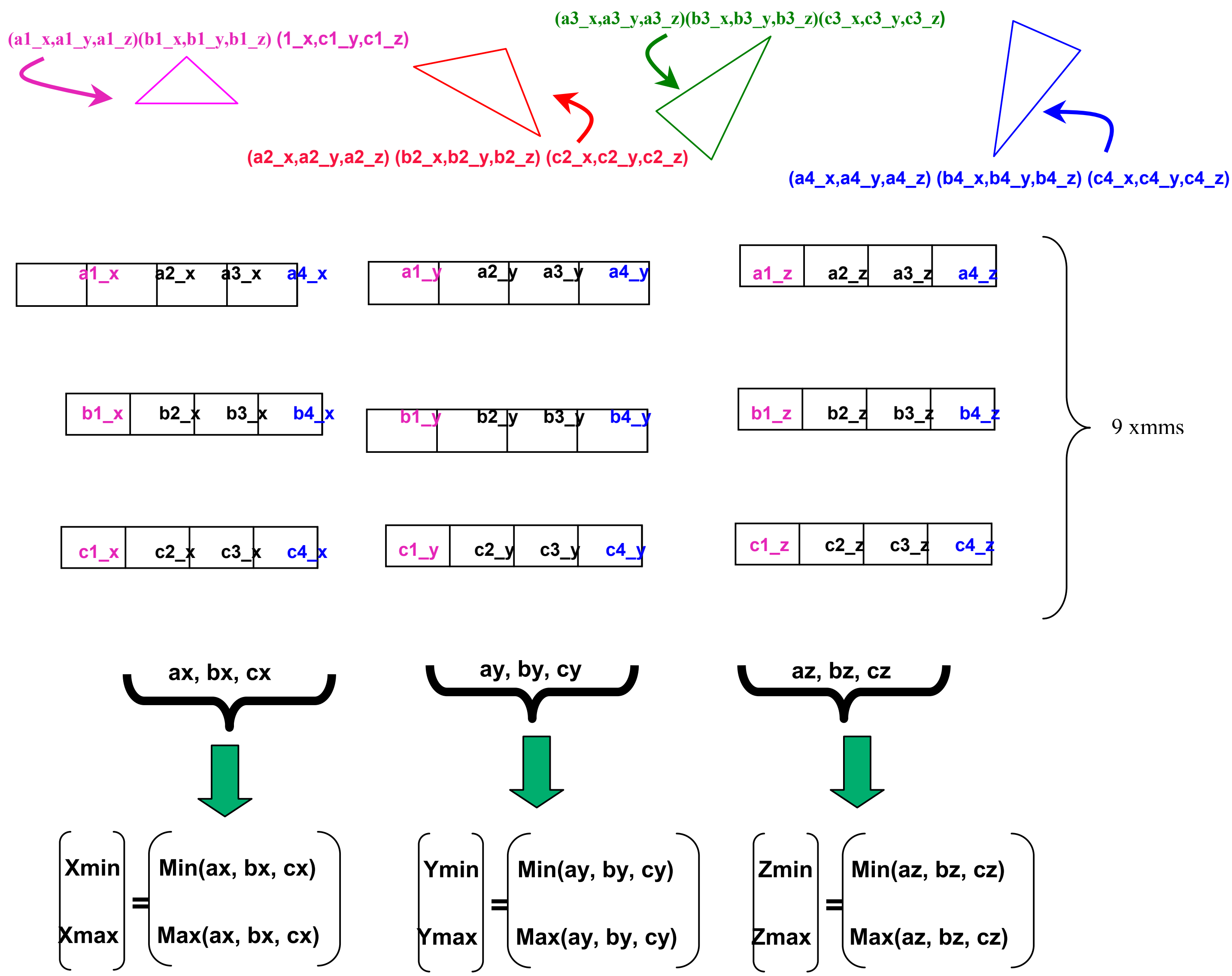

Figure 4 SSE2 Four triangles at a time

Conceptually, and as shown in Figure 4, only nine XMM registers are needed to accommodate the data for 4 triangles rather than 12 XMM registers in the earlier approach. This gives an insight into why dealing with four triangles at a time would be beneficial: it allows use of the "register" estate at our disposal better (no slots wasted in the XMM registers). This changes the way the computation for the bounding boxes are performed. In Figure 4, **ax** can be thought of as a vector that has assembled the x-components of the first point (the "a" point) of each of the four triangles. Similarly **bx** is the vector assembling the x-components of the second point in each of the four triangles (the "b" point) and **cx** is the vector assembling the x components of the third point of each of the four triangles (the "c" point). Similar interpretations hold for vectors **ay,by,cy,az,bz** and **cz**). **Xmin** results from the component-wise minimum of the vectors **ax**, **bx** and **cx** and thus each component of **Xmin** is the minimum x-component of each of the four triangles. Similar interpretation holds for all the other vectors, both min and max. The result is six XMM registers that have the complete bounding box data of 4 triangles.

The number of XMM registers needed by the procedure described above exceeds the number of those available on the current Pentium 4 processor. To keep the XMM register count within the number of physically available registers, computations are done in individual coordinate directions, one at a time (i.e., x-coordinate computations are done first followed by computations with the y and z coordinates). Since the operations in the three coordinate directions are independent of each other the operations can be performed in any order. The increased overhead from repetitive memory reads is offset by the increase in performance resulting from working with four triangles at a time.

The steps of converting to integers and clamping are similar to the ones described in the previous section. The only difference is that every time these operations are performed, all slots in the XMM registers are utilized, indicating more efficient use. Figure 6 shows how the first and last bounding boxes are assembled from the six XMM registers.

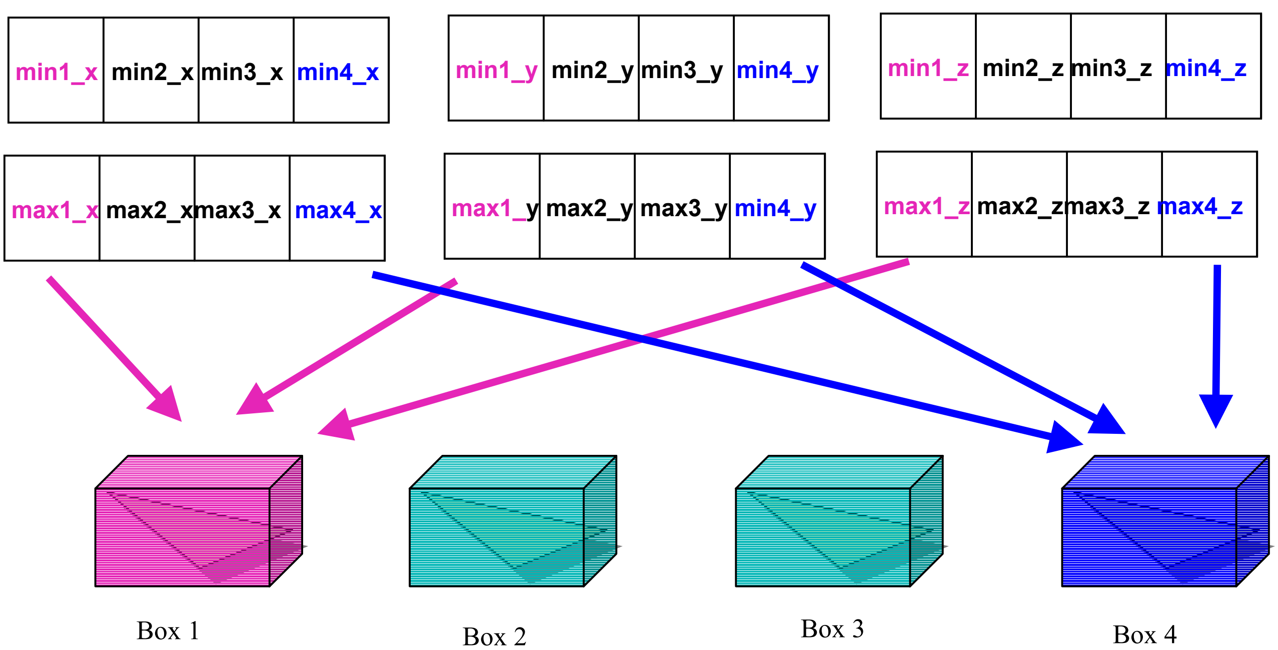

Figure 5 Bounding Box output stream

Relative performance, one triangle vs. four triangles at a time

This section outlines results of processing triangle data using methods developed in the previous sections. The results below are for 2.5 million triangles generated randomly and processed to compute their bounding boxes. The results show a 23 % improvement in performance for an OpenGL triangle data stream when the processing is done four triangles at a time vs. one triangle at a time (note both are SSE2 implementations).

System	Time in seconds
P4 1.7 GHz – SSE2 (four triangles)	0.24747 sec
P4 1.7 GHz (one triangle)	0.3045sec

Table 1 - Performance of four triangles vs. one triangle at a time using SSE2

OpenGL data in a tri-strip format

OpenGL can often be instructed to arrange triangle data in a tri-strip format. The advantage of this format is that the amount of data needed to describe the triangles is minimized. This section examines an approach similar in theme to the last section. A tri-strip is constructed by representing the starting triangle with all three vertices, but for each additional triangle (which shares an edge with the triangle), only the third (new) vertex is stored. Thus for 2 triangles 4 vertices are stored. In general for N triangles that can be represented in a tri-strip N+2 vertices are stored. Figure 7 shows the construction of a tri-strip. Note the change in the vertex naming scheme. It is easy to remember which new vertex is describing the next triangle, for example the vertex with the suffix 7 in the coordinates represents the 6^{th} triangle in the tri-strip.

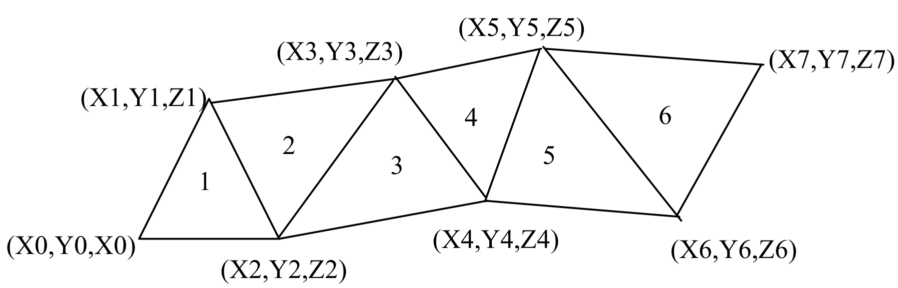

Figure 6 - Tri-Strip arrangement

The first 4 triangles in the tri-strip shown in Figure 6 would be represented in the stream below.

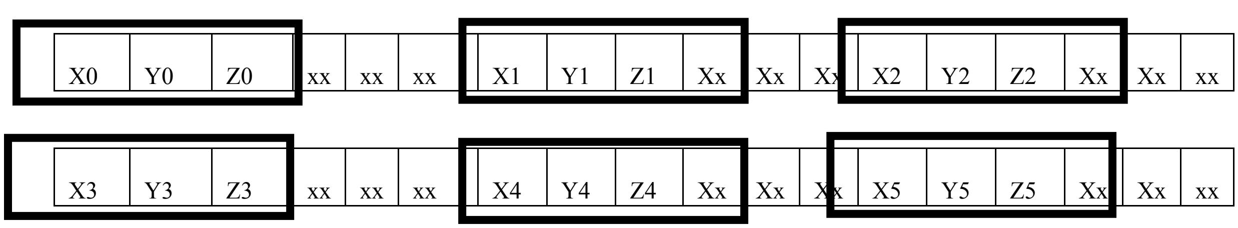

Figure 7 Tri-strip OpenGL data stream

The triangles themselves are the following triads:

Triangle 1 : (X0,Y0,Z0),(X1,Y1,Z1),(X2,Y2,Z2)

Triangle 2 : (X1,Y1,Z1),(X2,Y2,Z2),(X3,Y3,Z3)

Triangle 3: (X2,Y2,Z2),(X3,Y3,Z3),(X4,Y4,Z4)

Triangle 4 : (X3,Y3,Z3),(X4,Y4,Z4),(X5,Y5,Z5)

In this case, reading the five points from memory would complete all data required to construct the four triangles in XMM registers. Then a series of shuffles, masks, ANDs and ORs are performed to get the data in the format that is familiar from the last section. Once the first vector (xmm0) is completed, the next one is just a shift to the right and a shuffle of X4 (which comes from the next sequential point in the data stream) in the high order slot in the register. To get the minimum and maximum extents of the four triangles, two min() operations and two max() operations are needed. The process is illustrated for the x-components in the Figure 9 below.

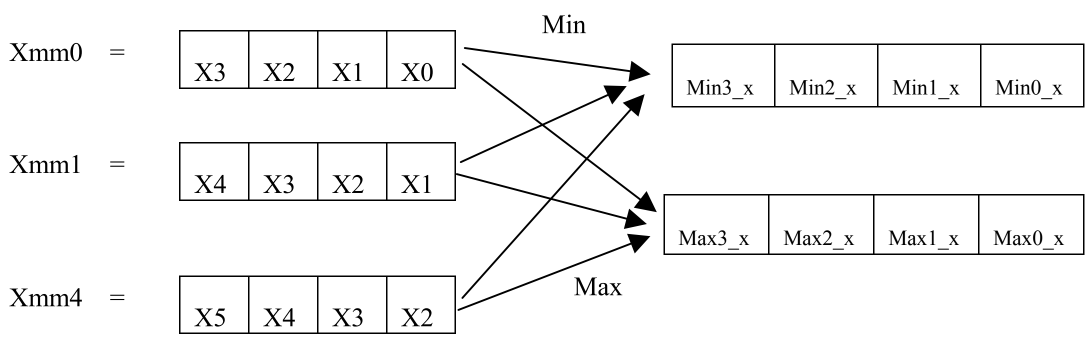

Figure 8 Three way min and max for 4 triangles at a time (showing x-coordinate)

The y-component and z-components can be handled similarly. The process of assembling the bounding boxes has been explained in detail in the previous sections.

Relative performance, tri-strips vs. disjoint triangles

It is obvious that the tri-strips method holds an advantage because there is lesser data involved, requiring fewer memory reads. In similar experiments as described in the previous sections (processing 2.5 million valid triangles) the SSE2 implementation that is developed for the tri-strip method of input data is approximately 20 % faster than the case for disjoint triangles. The improvement in performance is perhaps not as dramatic as would intuitively be expected (since the number of memory accesses are actually halved) because there is increased overhead in arranging the tri-strip data into appropriate XMM registers.

System	**Time in seconds**
P4 1.7 GHz – SSE2 (four triangles)	0.24747 sec

P4 1.7 GHz (one triangle)	0.3045 sec
P4 1.7 GHz (tri-strips, four triangles)	0.2013 sec

Summary

This paper is meant to be instructional in the use of Streaming Single Instruction Multiple Data Extensions (SSE) in parallelizing computations in real world applications. An OpenGL data stream of triangles is considered and two different ways of parallelizing the computation of bounding boxes are investigated. It is explained how, even though both approaches are significantly better than non-SSE approaches, one approach outperforms that other because of more complete use of SSE2 register resources. Performance is compared on an appropriate workload set of triangles.

Appendix A

SSE2 assembly code snippet for processing 4 triangles at a time

```
movlps      xmm2, [eax+0]
movhps      xmm2, [eax+6*4*3]          // xmm2: a1y,a1x,a0y,a0x
movlps      xmm3, [eax+6*4*3*2]
movhps      xmm3, [eax+6*4*3*3]       // xmm3: a3y,a3x,a2y,a2x
shufps      xmm2,xmm3,0x88            // xmm2=a3x,a2x,a1x,a0x (**)

movlps      xmm3, [eax+6*4]
movhps      xmm3, [eax+6*4*3+6*4]      // xmm3: b1y,b1x,b0y,b0x
movlps      xmm4, [eax+6*4*3*2+6*4]
movhps      xmm4, [eax+6*4*3*3+6*4]    // xmm4: b3y,b3x,b2y,b2x

shufps      xmm3,xmm4,0x88             // xmm3=b3x,b2x,b1x,b0x (**)


movlps      xmm4, [eax+6*4*2]
movhps      xmm4, [eax+6*4*3+6*4*2]    // xmm4: c1y,c1x,c0y,c0x
movlps      xmm5, [eax+6*4*3*2+6*4*2]
movhps      xmm5, [eax+6*4*3*3+6*4*2]  // xmm5: c3y,c3x,c2y,c2x
shufps      xmm4,xmm5,0x88             // xmm4=c3x,c2x,c1x,c0x (**)

movaps      xmm5, xmm2
minps       xmm2, xmm3                 // xmm2: min x(not final)
maxps       xmm5, xmm3                 // xmm5: max x(not final)
minps       xmm2, xmm4                 // xmm2: min x-(final) (**)
maxps       xmm5, xmm4                 // xmm5: max x-(final) (**)

// Clamp all values onto [0.0 , 1023,0]
movups         xmm3,[fOnekm]          // xmm3: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4              // xmm4: 0 | 0 | 0 | 0
maxps          xmm2,xmm4
maxps          xmm5,xmm4
```

```
minps          xmm2,xmm3            // xmm2: qfmin3x | qfmin2x | qfmin1x | qfmin0x

minps          xmm5,xmm3            // xmm5: qfmax3x | qfmax2x | qfmax1x | qfmax0x

cvttps2dq   xmm2,xmm2
cvttps2dq   xmm5,xmm5

xorps       xmm6,xmm6           //clear
xorps       xmm7,xmm7           //clear
por         xmm6,xmm2           //xmm6: minx3|minx2|minx1|minx0
por         xmm7,xmm5           //xmm7: max3 | max2| max1| max0

// do the y coord

movlps      xmm2, [eax+0]
movhps      xmm2, [eax+6*4*3]          // xmm2: a1y,a1x,a0y,a0x
movlps      xmm3, [eax+6*4*3*2]
movhps      xmm3, [eax+6*4*3*3]       // xmm3: a3y,a3x,a2y,a2x
shufps      xmm2,xmm3,0xdd            // xmm2=a3y,a2y,a1y,a0y (**)

movlps      xmm3, [eax+6*4]
movhps      xmm3, [eax+6*4*3+6*4]      // xmm3: b1y,b1x,b0y,b0x
movlps      xmm4, [eax+6*4*3*2+6*4]
movhps      xmm4, [eax+6*4*3*3+6*4]    // xmm4: b3y,b3x,b2y,b2x
shufps      xmm3,xmm4,0xdd             // xmm3=b3y,b2y,b1y,b0y (**)

movlps      xmm4, [eax+6*4*2]
movhps      xmm4, [eax+6*4*3+6*4*2]    // xmm4: c1y,c1x,c0y,c0x
movlps      xmm5, [eax+6*4*3*2+6*4*2]
movhps      xmm5, [eax+6*4*3*3+6*4*2]  // xmm5: c3y,c3x,c2y,c2x
shufps      xmm4,xmm5,0xdd             // xmm4=c3y,c2y,c1y,c0y (**)

movaps      xmm5, xmm2
minps       xmm2, xmm3                 // xmm2: min y(not final)
maxps       xmm5, xmm3                 // xmm5: max y(not final)
minps       xmm2, xmm4                 // xmm2: min y-(final) (**)
maxps       xmm5, xmm4                 // xmm5: max y-(final) (**)

// Clamp all values onto [0.0 , 1023,0]
movups         xmm3,[fOnekm]          // xmm3: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4              // xmm4: 0 | 0 | 0 | 0
maxps          xmm2,xmm4
maxps          xmm5,xmm4
minps          xmm2,xmm3            // xmm2: qfmin3y | qfmin2y | qfmin1y | qfmin0y
minps          xmm5,xmm3           // xmm5: qfmax3y | qfmax2y | qfmax1y | qfmax0y

cvttps2dq   xmm2,xmm2
cvttps2dq   xmm5,xmm5

pslld       xmm2,10
```

```
pslld       xmm5,10

por         xmm6,xmm2     //xmm6: miny3,minx3|miny2,minx2|miny1,minx1|miny0,minx0
por         xmm7,xmm5     //xmm7: maxy3,max3 | maxy2,max2| maxy1,max1| maxy0,max0

// do the z coord
movlps      xmm2, [eax+4*2]
movhps      xmm2, [eax+6*4*3+4*2]          // xmm2: ~,a1z,~,a0z

movlps      xmm3, [eax+6*4*3*2+4*2]
movhps      xmm3, [eax+6*4*3*3+4*2]        // xmm3: ~,a3z,~,a2z
shufps      xmm2,xmm3,0x88                 // xmm2=a3z,a2z,a1z,a0z (**)

movlps      xmm3, [eax+6*4+4*2]
movhps      xmm3, [eax+6*4*3+6*4+4*2]      // xmm3:~,b1z,~,b0z
movlps      xmm4, [eax+6*4*3*2+6*4+4*2]
movhps      xmm4, [eax+6*4*3*3+6*4+4*2]    // xmm4:~,b3z,~b2z
shufps      xmm3, xmm4,0x88                // xmm3:b3z,b2z,b1z,b0z (**)

movlps      xmm4, [eax+6*4*2+4*2]
movhps      xmm4, [eax+6*4*3+6*4*2+4*2]    // xmm4: ~,c1z,~,c0z
movlps      xmm5, [eax+6*4*3*2+6*4*2+4*2]
movhps      xmm5, [eax+6*4*3*3+6*4*2+4*2]  // xmm5: ~,c3z,~,c2z
shufps      xmm4,xmm5,0x88                 // xmm4=c3z,c2z,c1z,c0z (**)

movaps      xmm5, xmm2
minps       xmm2, xmm3                 // xmm2: min z(not final)
maxps       xmm5, xmm3                 // xmm5: max z(not final)
minps       xmm2, xmm4                 // xmm2: min z-(final) (**)
maxps       xmm5, xmm4                 // xmm5: max z-(final) (**)

// Clamp all values onto [0.0 , 1023,0]
movups         xmm3,[fOnekm]          // xmm3: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4             // xmm4: 0 | 0 | 0 | 0
maxps          xmm2,xmm4

maxps          xmm5,xmm4
minps          xmm2,xmm3      // xmm2: qfmin3z | qfmin2z | qfmin1z | qfmin0z
minps          xmm5,xmm3      // xmm5: qfmax3z | qfmax2z | qfmax1z | qfmax0z

cvttps2dq   xmm2,xmm2
cvttps2dq   xmm5,xmm5

pslld       xmm2,20
pslld       xmm5,20
por         xmm6,xmm2
//xmm6: minz3,miny3,minx3|minz2,miny2,minx2|minz1,miny1,minx1|minz0,miny0,minx0
por         xmm7,xmm5
//xmm7: maxz3,maxy3,max3 | maxz2,maxy2,max2| maxz1,maxy1,max1| maxz0,maxy0,max0
```

```
movaps      xmm2,xmm6

punpckldq   xmm6,xmm7           //xmm6: first 2 triangles max1|min1|max0|min0
punpckhdq   xmm2,xmm7           //xmm2: next  2 triangles max3|min3|max2|min2

//ship them out
movdqu     [ebx],xmm6
movdqu     [ebx+16],xmm2
```

Appendix B

SSE2 code snippet for tri-strips OpenGL data stream

```
movups      xmm0,[eax]          // xmm0: xx,z0,y0,x0
movups      xmm1,[eax+6*4]      // xmm1: xx,z1,y1,x1
movups      xmm2,[eax+6*4*2]    // xmm2: xx,z2,y2,x2
movups      xmm3,[eax+6*4*3]    // xmm3: xx,z3,y3,x3
movups      xmm4,[eax+6*4*4]    // xmm4: xx,z4,y4,x4
movups      xmm5,[eax+6*4*5]    // xmm5: xx,z5,y5,x5

pshufd    xmm0,xmm0,0x10        // xmm0: xx,y0,xx,x0
pshufd    xmm1,xmm1,0x40        // xmm1: y1,xx,x1,xx
movaps    xmm6,[mask1]          // xmm6: 0xffffff,0,0xffffff,0
pand      xmm1,xmm6             // xmm1: y1,0,x1,0
psrldq    xmm6,4                // xmm6: 0x0,0xffffffff,0,0xffffffff,
pand      xmm0,xmm6             // xmm0:  0,y0, 0,x0
por       xmm0,xmm1             // xmm0: y1,y0,x1,x0

pshufd    xmm2,xmm2,0x10        // xmm2: xx,y2,xx,x2
pshufd    xmm3,xmm3,0x40        // xmm3: y3,xx,x3,xx
pand      xmm2,xmm6             // xmm2:  0,y2, 0,x2
pslldq    xmm6,4                // xmm6:0xffffff,0,0xffffff,0
pand      xmm3,xmm6             // xmm3: y3, 0,x3, 0
por       xmm2,xmm3             // xmm2: y3,y2,x3,x2

movaps      xmm6, xmm0          // xmm6 = xmm0
movaps      xmm7, xmm2          // xmm7 = xmm2
shufpd      xmm0,xmm7,0         // ** xmm0: x3,x2,x1,x0

psrldq     xmm6,8               // xmm6:0,0,y1,y0   -- shift right by bytes
shufpd     xmm6,xmm2,2          // xmm6:y3,y2,y1,y0
movaps     xmm2, xmm6           //** xmm2: y3,y2,y1,y0

movaps      xmm1, xmm0          // xmm1 = xmm0
movss       xmm1, xmm4          // xmm1: x3,x2,x1,x4
pshufd      xmm1, xmm1,0x39     // ** xmm1: x4,x3,x2,x1
movaps      xmm3, xmm2          // xmm3 = xmm2
psrldq      xmm4, 4             // xmm4: 00,xx,z4,y4  // dont need x4 anymore
movss       xmm3, xmm4          // xmm3: y3,y2,y1,y4
pshufd      xmm3, xmm3,0x39     // ** xmm3: y4,y3,y2,y1
```

```
movaps      xmm4,xmm1           // xmm4 = xmm1

movss       xmm4, xmm5          // xmm4: x4,x3,x2,x5
pshufd      xmm4,xmm4,0x39      // **xmm4: x5,x4,x3,x2
psrldq      xmm5, 4             // xmm5: 00,xx,z5,y5 // dont need x5 anymore

movaps      xmm7, xmm3          // xmm7 = xmm3
movss       xmm7,xmm5           // xmm7:  y4,y3,y2,y5
pshufd      xmm7,xmm7,0x39      // xmm7:  y5,y4,y3,y2
movaps      xmm5, xmm7          // **xmm5: y5,y4,y3,y2

movaps xmm6, xmm0
minps  xmm0, xmm1
minps  xmm0, xmm4               // **xmm0: minx3,minx2,minx1,minx0
maxps  xmm1, xmm6
maxps  xmm1, xmm4               // **xmm1: maxx3, maxx2, maxx1, maxx0

movaps xmm6, xmm2
minps  xmm2, xmm3
minps  xmm2, xmm5               // **xmm2: miny3,miny2,miny1,miny0
maxps  xmm3, xmm6
maxps  xmm3, xmm5               // **xmm3: maxy3,maxy2,maxy1,maxy0

// Clamp all values onto [0.0 , 1023,0]
movups         xmm7,[fOnekm]         // xmm7: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4             // xmm4: 0 | 0 | 0 | 0
maxps          xmm0,xmm4
maxps          xmm1,xmm4
minps          xmm0,xmm7      // **xmm0: qfmin3x | qfmin2x | qfmin1x | qfmin0x
minps          xmm1,xmm7      // **xmm1: qfmax3x | qfmax2x | qfmax1x | qfmax0x

cvttps2dq   xmm0,xmm0           // SSE2
cvttps2dq   xmm1,xmm1           // SSE2

xorps       xmm6,xmm6           //clear
xorps       xmm7,xmm7           //clear
por         xmm6,xmm0           //xmm6: minx3|minx2|minx1|minx0
por         xmm7,xmm1           //xmm7: maxx3|maxx2|maxx1|maxx0

// Clamp all values onto [0.0 , 1023,0]
movups         xmm5,[fOnekm]  // xmm5: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4      // xmm4: 0 | 0 | 0 | 0
maxps          xmm2,xmm4
maxps          xmm3,xmm4
minps          xmm2,xmm5      // xmm2: qfmin3y | qfmin2y | qfmin1y | qfmin0y
minps          xmm3,xmm5      // xmm3: qfmax3y | qfmax2y | qfmax1y | qfmax0y

cvttps2dq   xmm2,xmm2           // SSE2
cvttps2dq   xmm3,xmm3           // SSE2
```

```
pslld       xmm2,10

pslld       xmm3,10

//xmm6: miny3,minx3|miny2,minx2|miny1,minx1|miny0,minx0
por         xmm6,xmm2
//xmm7: maxy3,maxx3|maxy2,maxx2|maxy1,maxx1|maxy0,maxx0

por         xmm7,xmm3

// z's
// have to perform the same fetches again but they should be in the cache
movups      xmm0,[eax+2*4]              // xmm0: xx,xx,xx,z0
movups      xmm1,[eax+6*4 + 2*4]        // xmm1: xx,xx,xx,z1
movups      xmm2,[eax+6*4*2 + 2*4]      // xmm2: xx,xx,xx,z2
movups      xmm3,[eax+6*4*3 + 2*4]      // xmm3: xx,xx,xx,z3
movups      xmm4,[eax+6*4*4 + 2*4]      // xmm4: xx,xx,xx,z4

// load a mask and mask everything:
movaps  xmm5, [mask]
pand xmm0, xmm5                          // xmm0: 0,0,0,z0
pand xmm1, xmm5                          // xmm1: 0,0,0,z1
pand xmm2, xmm5                          // xmm2: 0,0,0,z2
pand xmm3, xmm5                          // xmm3: 0,0,0,z3
pand xmm4, xmm5                          // xmm4: 0,0,0,z4

pslldq xmm3, 0xc                        // xmm3: z3, 0, 0, 0
pslldq xmm1, 0x4                        // xmm1:  0, 0,z1, 0
pslldq xmm2, 0x8                        // xmm2:  0,z2, 0, 0
por xmm0,xmm3
por xmm0,xmm1
por xmm0,xmm2                           // ** xmm0: z3,z2,z1,z0

// do point 6
movaps      xmm1, xmm5                  // copy mask
movups      xmm5,[eax+6*4*5 + 2*4]      // xmm5: xx,xx,xx,z5
pand   xmm5,xmm1                        // xmm5: 0,0,0,z5

movaps xmm1, xmm0                       // xmm1 = xmm0
movss xmm1,xmm4                         // xmm1: z3,z2,z1,z4
pshufd xmm1,xmm1,0x39                   // ** xmm1: z4,z3,z2,z1
movaps xmm2,xmm1                        // xmm2 = xmm1
movss xmm2,xmm5                         // xmm2: z4,z3,z2,z5
pshufd xmm2,xmm2,0x39                   // ** xmm2: z5,z4,z3,z2

movaps xmm3, xmm0
minps xmm0,xmm1
minps xmm0,xmm2                         // ** xmm0: minz3,minz2,minz1,minz0
maxps xmm1,xmm3
maxps xmm1,xmm2                         // ** xmm1: maxz3,maxz2,maxz1,maxz0
```

```
// Clamp all values onto [0.0 , 1023,0]
movups         xmm3,[fOnekm]         // xmm3: 1023 | 1023 | 1023 | 1023
xorps          xmm4,xmm4             // xmm4: 0 | 0 | 0 | 0
maxps          xmm0,xmm4
maxps          xmm1,xmm4
minps          xmm0,xmm3      // xmm0: qfmin3z | qfmin2z | qfmin1z | qfmin0z
minps          xmm1,xmm3      // xmm1: qfmaxx3z | qfmax2z | qfmax1z | qfmax0z

cvttps2dq   xmm0,xmm0           // SSE2
cvttps2dq   xmm1,xmm1           // SSE2

pslld       xmm0,20
pslld       xmm1,20
//xmm6: minz3,miny3,minx3|minz2,miny2,minx2|minz1,miny1,minx1|minz0,miny0,minx0
por         xmm6,xmm0
//xmm7: maxz3,maxy3,maxx3|maxz2,maxy2,maxx2|maxz1,maxy1,maxx1|maxz0,maxy0,maxx0
por         xmm7,xmm1

movaps      xmm2,xmm6
punpckldq   xmm6,xmm7           //xmm7: first 2 triangles max1|min1|max0|min0
punpckhdq   xmm2,xmm7           //xmm2: next  2 triangles max3|min3|max2|min2

//ship them out to memory
movdqu     [ebx],xmm6
movdqu     [ebx+16],xmm2
```