An Architecture for Ray - Bezier patch intersection

Peter De Vijt*, Luc Claesen†, Hugo De Man‡
IMEC, Kapeldreef 75, 3001 Heverlee, Belgium
e-mail: devijt@imec.be

Abstract

A new fast ray - patch intersection algorithm is presented. The algorithm correctly handles all ray - patch intersections. A number of parameters are derived from a numerical analysis of the algorithm and the data pad is re synthesized for higher accuracy. A global architecture for an ASIC for intersecting a ray with a bezier patch is presented. It is shown that a cache combined with pre pads can reduce the required memory considerable with an extremely small performance penalty. Attention will be paid to the scheduling and control problem. Several high level optimizations are presented that make efficient scheduling possible and decrease the calculation time considerably.

1 Introduction

In recent years there has been a lot of interest for direct ray patch intersection calculation in hardware. A number of architectures have been proposed [PK87a] [PK87b] [BSC89] [Sch87] [vdV89]. These papers mainly concentrate on the data path, and few or no details of the control and scheduling algorithm are presented. A number of problems, due to the recursive nature of the algorithm, still exist.

Since the implementation of the different sub blocks of the data pad is straightforward, not much detail will be given. Instead more attention will be paid to the numerical aspects, the data management and the scheduling problem. Some high level optimizations, that result in a better schedule and faster execution will be presented. Several errors in previous implementations will be corrected.

*Supported by a grant from IWONL
†IMEC, Professor at KUL
‡IMEC, Professor at KUL

2 Overview of intersection algorithms

The intersection calculation of a ray with a patch can be done in several ways:

- **Numerically** with interval Newton iteration [Tot85], quasi Newton optimization [JB86] and control mesh refinement followed by a Newton iteration [SB86] [LTM87].
- **Algebraically** with a resultants method [Kaj82].
- Also **subdivision** methods can be used in three [Cla79] [LCWB80] [LR80] [Yan87] or two dimensions [Rog85] [Woo89]. [NSK90] also handles trimmed rational patches with an adaptive subdivision algorithm.

For a hardware implementation an algorithm with a highly repetitive inner loop and not too much exceptions is needed. Only the simpler subdivision algorithms seem to be suited. A number of approaches have been tested based on the three dimensional subdivision algorithm for a number of patches: [PK87a] [PK87b] [BSC89] describe architectures for bezier patches, [Sch87] [vdV89] describe architectures for rational B-splines.

3 The subdivision algorithm

3.1 the algo

The algorithm that was chosen as a basis is the same as in [PK87a] [PK87b]. The patches are scanned in a depth first order. The following algorithm is executed on every patch on the stack:

- pop patch from stack and subdivide
- determine bounding boxes for the sub patches
- calculate the intersection point of the ray with these bounding boxes
- sort them
- if there is a hit
  - if termination requirement met
    - if there was an intersection compare if the new one is closer
    - otherwise it is the first intersection
  - else push the sub patch on the stack

This algorithm only serves as a basis, several optimizations will be presented in the following sections.

3.2 Inverse direction

The algorithm requires 3 divisions per sub patch to calculate the intersection of the ray with the bounding box. Since these factors stay the same for a given ray throughout all the recursions, they can be stored and reused. Previous work did however not mention how these factors were actually calculated. Since an division requires an important amount of area and time, this has an impact on the implementation.

These factors can however be derived without divisions. For a given ray direction $\overline{d} = (a, b, c)$ the vector $\overline{d'} = (1/a, 1/b, 1/c)$ is to be calculated.. The vector $\overline{d'}$ and the vector $s.\overline{d'}$ represent the same direction. A constant scaling factor $s = a.b.c$ can be chosen. The vector $s.\overline{d'}$ then becomes $s.\overline{d'} = (b.c, a.c, a.b)$ which can be calculated with multiplications only.

3.3 Iteration depth

The parameter space is halved every iteration and the real world space is approximately halved. The more the parameter space is uniform the more this approximation holds. The accuracy needed for the intersection point depends on the size of the projected patch on the screen. Most authors use however a fixed accuracy. Different authors propose a different required accuracy: For a 512 by 512 resolution [LTM87] and [Yan87] propose 2^{-6}, [Tot85] and [NSK90] propose 2^{-10}, [BSC89] and [Woo89] propose 2^{-12}, [JB86] proposes 2^{-16} and [PK87a] [PK87b] propose 2^{-9} for display purposes and 2^{-21} for high accuracy solid modelling. The different authors do not agree well.

From the results of some experiments we estimate that for a resulting sub patch to be sub pixel resolution about 2^{-10} for a 512 by 512 image and 2^{-12} for a 2k by 2k image is needed. If super sampling or stochastic sampling is used, a factor 2^{-4} on top of this is needed. The total accuracy needed ranges thus from 2^{-10} to 2^{-16}. From 10 to 16 subdivisions are needed.

4 Numerical analysis

4.1 Number representation

The number representation should satisfy two conditions: (i) small implementation possible and (ii) high accuracy of the results. A small implementation dictates the use of integer arithmetic. A transformation will be used to satisfy the second condition.

From the convex hull property of bezier curves it follows that the range of the control points of the sub patches is bounded. This makes it possible to transform the control points to a representation that is has a limited range such as an integer representation. Several transformations are possible. The transformation $[\ a \quad b\ ] \rightarrow [\ 0 \quad (b-a)2^{-n}\ ]$ seems a good compromise between

optimal filling of the transformed data space and ease of transformation (maximally one bit is spoiled and only addition and scaling with power of 2 are needed)

If the relation $b > 2^m(b-a)$ with $m \geq 1$ holds the n bit integer representation will have the same accuracy as an $n+m$ bit mantissa floating point representation. (since the floating point representation will have m leading 1's). Only in the cases where $m < 1$ the floating point representation will be more accurate for a part of the interval [a $b/2$] and will have the same accuracy for the interval [$b/2$ b].

The patches can be scaled during a pre processing step, the ray has to be transformed during the execution phase. In contrast to [PK87a] [PK87b] [BSC89] the ray is translated to the bounding box origin only if the ray origin was outside of this bounding box, otherwise not all cases (e.g. reflected rays) can be handled correctly.

4.2 Error propagation

The algorithm recursively subdivides patches until a stop criterium is met. Since the subdivision section uses additions and down shifts a truncation error will be made. Every iteration this error is propagated and a new error is generated. It is possible to reduce the total error by reducing the number of truncations. This can be accomplished by shifting down only after a number of additions instead of after each addition. If first addition is done with n bits, the next one will have to be done with $n+1$ bits and so on. It seems that the price that has to be paid for decreasing the error is an increase in area. A further analysis shows that this not the case. The truncation error ϵ for shifting down is

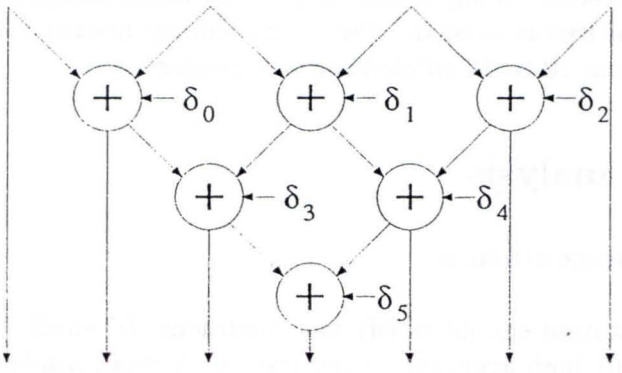

Figure 1: The curve subdivision data pad

$$\lfloor \frac{a}{2^n} \rfloor \rightarrow \varepsilon \epsilon [\ 0 \quad \frac{2^n-1}{2^n}\ ]$$

It is clear that when the result of every operation is rounded down, every iteration the control points will drift down. If a constant is added to the number

before shifting down the error is:

$$\lfloor \frac{a+\delta}{2^n} \rfloor \; with \; \delta < 2^n \;\rightarrow\; \varepsilon \in [\; -\tfrac{\delta}{2^n} \quad \tfrac{2^n-1-\delta}{2^n} \;]$$

The bounds are equally centered around 0 if δ satisfies:

$$\begin{cases} \delta = 2^{n-1} - 1 & \rightarrow \quad \varepsilon \in [\; -\frac{2^{n-1}-1}{2^n} \quad \frac{1}{2} \;] \\ \delta = 2^{n-1} & \rightarrow \quad \varepsilon \in [\; -\frac{1}{2} \quad \frac{2^{n-1}-1}{2^n} \;] \end{cases}$$

Every iteration the error grows due to propagation and generation:

$$\varepsilon_{i+1} = \varepsilon_i p_i + g_i$$

The error range after 1 and 16 iterations for truncating and rounding for the different configurations is:

iterations	after	truncate	round
1	1	[0.00 1.50]	-
	3	[0.00 0.87]	[−0.50 0.37]
	6	[0.00 0.98]	[−0.50 0.42]
16	1	[0.00 16.33]	-
	3	[0.00 11.33]	[−6.06 5.48]
	6	[0.00 10.33]	[−5.48 4.95]

The rounding after 3 adders has about two bits more accuracy than the truncation after one adder. The word length of the version after 3 adders can be two bits less than the version after 1 adder.

The following table summarizes the cost for the subdivision of a curve with n bit control points when the shift down is done after a different number of adders. The generic case is the cost for an n bit implementation, the compensated case takes the required accuracy into account:

after	generic		compensated	
	FA	mux	FA	mux
1	6n	6n	-	-
3	6n+4	6n	6n-8	6n-8
6	6n+13	6n+6	6n+1	6n-6

As can be seen from the table, the case after 3 adders is the best. Since the number of bits is smaller than the original case, there is also a reduction in the required memory. Note that for a n bit implementation, the intersection point can only be calculated with n-2 bits of accuracy.

Instead of using additional adders for rounding, the appropriate δ can be injected in the adder network by using the carry input. A different δ is needed for each output. A set of δ_i's have to be chosen so that the rounding can be performed for all outputs.

The sets (1,1,1,0,0,0), (1,1,1,0,0,1) and (1,0,1,1,1,0) satisfy these conditions for the case after 3 adders. For the case after 6 adders, there is no set that satisfies the conditions, and additional adders would be needed.

In a bit serial implementation as in [PK87a] [PK87b] the subdivision can be done exactly in a subdivision chain at the cost of a reduced pipeline efficiency since pipeline bubbles are inserted. The subdivision in stages however introduces errors since a fixed number of bytes is saved. [PK87a] [PK87b] don't account for this.

4.3 Normal calculation

The normal will be calculated as a product of the tangents in the u and v direction.

The tangent of a curve

$$C(t) = \sum_{i=0}^{3} B_{i,3}(t) P_i$$

is

$$C'(t) = \sum_{i=0}^{3} B'_{i,3}(t) P_i$$

Evaluated in $t = 1/2$ this becomes

$$C'(t)|_{1/2} = 3/4(-P_0 - P_1 + P_2 + P_3)$$

Since only in the direction is needed and not the length, the factor 3/4 can be dropped. If the patch is flat enough this can be further reduced to

$$C'(t)|_{1/2} \sim -P_0 + P_3$$

Since a subtraction of two almost equal numbers is done during the calculation of $C'(t)|_{1/2}$ a catastrophic cancellation can occur. This can be avoided by making sure $|-P_0 + P_3|$ is bounded by a lower limit so that the relative error on $C'(t)|_{1/2}$ has an upper bound. The relative error is then given by $\varepsilon_r = |-P_0 + P_3|^{-1}$.

The tangents need to be known with relative error $\varepsilon_{rt} = \varepsilon_{rn}/2$ for the normal to be known with relative error ε_{rn} This means that the minimal size of the bounding box for the patch be $|-P_0 + P_3| > 2\varepsilon_{rn}^{-1}$. If the normal is to be known with a relative error of $\varepsilon_{rn} = 2^{-n}$ n+1 more bits are needed for the normal calculation than for the intersection calculation. The total number of bits required is thus $i+2+n+1$ for a relative error $\varepsilon_i = 2^{-i}$ on the intersection point and an relative error $\varepsilon_n = 2^{-n}$ on the normal. An implementation that calculates intersection point and normal needs a lot more bits than an implementation that only calculates the intersection. It will be shown in section 7.4 that this can be handled efficiently with multiple precision operators.

5 Architecture

5.1 The data pad

From some early experiments it was clear that a word parallel implementation could be integrated. Every cycle 4 words are dealt with in parallel. This means that every cycle a curve can be subdivided in the block Sub (see also fig. 1). The RC block is a block that transposes a 4 by 4 matrix (this is needed to operate on the 2 parameters u and v of the patch) The implementation of the data pad itself is straightforward. It is almost a one to one translation of the signal flow graph.

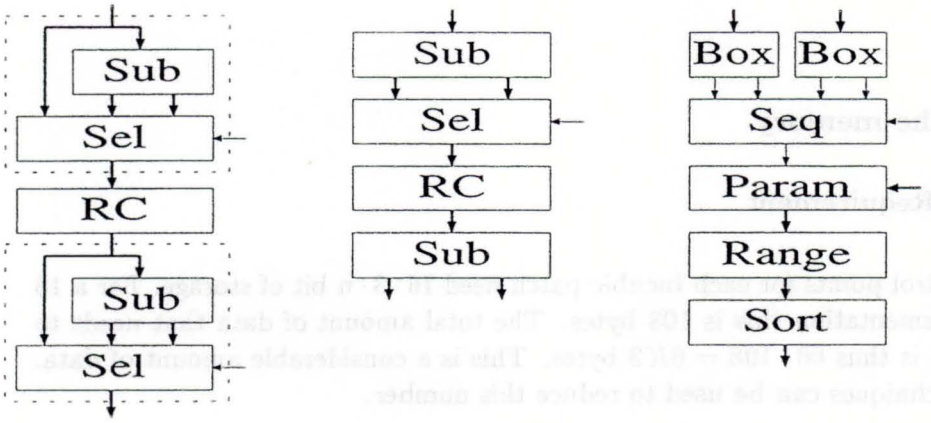

Figure 2: the complete data pad

Fig. 2 gives a general overview of the data pad. The left part is the pre processing pad. The operators are grouped according to their parameter space. The dashed sections can be repeated several times. The block consists out of a number of subdivision (Sub) and select (Sel) blocks for the first parameter, a transposition block (RC) and more subdivision and select blocks for the second parameter. (more on the reason for these pre pads in section 5.3) The output is fed to the core and the memory.

The middle part is the actual core. A curve is subdivided (Sub) in the first parameter, 1 out of 2 is selected (Sel), it is transposed (RC) and subdivided (Sub) in the second parameter.

The right part does the intersection check. A bounding box is calculated (Box), these values are fed to a sequencer (Seq), the parameter range is calculated for one dimension (Param), grouped for all dimensions (Range) and finally sorted according to the minimal distance. This information is put on the stack.

5.2 The stack

The information that is required for the next iteration are the data of the current patch and its path. In this design the stack holds the path definitions and pointers to the actual patch data (not the data itself). Since the maximum iteration depth that will be required is 16 and each level holds a maximum of 4 sub patches, a stack for $(16-1)\cdot 4 = 60$ places is needed. In theory only $(16-2)\cdot 3+4 = 46$ places are needed since for every iteration there are a maximum of 4 patches saved and one is read. Implementing this in silicon represents however some difficulties. [BSC89] uses a set of 12 stacks for a maximum subdivision level of 12.

At every level the 4 sub patches have almost the same path. The number of places on the stack can be reduced to 15 places and a set of valid bits for the 4 possible children when pre pads are used (more on this in the next section).

5.3 The memory

5.3.1 Requirement

The control points for each bicubic patch need $16\cdot 3\cdot n$ bit of storage. For a 18 bit implementation this is 108 bytes. The total amount of data that needs to be saved is thus $60\cdot 108 = 6K3$ bytes. This is a considerable amount of data. Some techniques can be used to reduce this number.

5.3.2 Port reduction

Each cycle 4 words need to be read and $4m$ words need to be stored, with m less or equal the number of sub patches generated per cycle (only the valid ones need to be saved). A problem is that at the moment the sub patches are generated, it is not known which ones need to be saved. In hardware 4 ports need to be provided. It is rather impossible to organize the memory so that the memory can be used efficiently. Room for 60 patches instead of 46 needs to be provided. This has to be provided for every independent patch that is needed to keep the pipeline filled (see also section 5.3.3)

To reduce the number of words that need to be stored and to be able to use the memory efficiently a pre processing pad is used to calculate one out of m sub patches. This way only the parent of the sub patches need to be stored. This can be organized efficiently. The technique of the pre pads is similar to what [PK87a] [PK87b] [BSC89] [Sch87] use for the reduction of the total memory (subdividing in stages). One pre pad reduces the input port requirement for the memory and the memory size with a factor 4.

5.3.3 Memory reduction

The memory can be further reduced with the same technique as for port reduction. With n additional pre pads the sub patch can be deduced from a patch n levels higher in the hierarchy. n additional pre pads reduce the memory requirement maximally with a factor n+1. The effect of the nth additional pre pad is only a (n+1)/n storage reduction. There is a trade off between the extra space needed for the data pad and the reduced memory. Since a word parallel implementation is used, there is no penalty as in [PK87a] [PK87b] for introducing pipeline bubbles.

The extra pre pads make the pipeline longer and impose extra burden on the scheduling. [PK87a] [PK87b] argument that a new sub patch can be scheduled before the outcome of the previous subdivision is known if the outcome is presented before the subdivision in the last pre pad is done. This is clearly not the case: for every backtrack the pipeline needs to be flushed with such a schedule. If k is the number of pre pads it is possible that the calculated and the needed sub patch have less than k-1 common branch directions. In this case the pipeline needs to be flushed.

Independent patches can be used to fill the pipeline. These independent patches need however additional storage. Introducing more pre pads does not necessarily decrease the memory requirement.

6 Cache Design

A cache is used to further reduce the total required memory. Since the access pattern of a stack is rather regular, a stack implemented with a cache can have a good performance. A modified LRU scheme has been used as a replacement policy. The patch with the but one lowest level will be replaced if the cache is full. In contrast with a normal cache implementation the replaced information will not be written to a memory lower in the hierarchy but will be simply discarded. This is done because otherwise a considerable amount of data would have to be transferred and the pipeline would have to be halted in the mean time. Data that is overwritten will later be recalculated from the lowest level patch available when needed.

To test the performance of the cache, access traces for the subdivision algorithm for 6 different data sets with 27 light sources were generated. The effect of some parameters was studied

6.1 Cache size

Fig. 3 shows the influence of the cache size on the relative execution speed for different data sets. The more patches that can be stored, the closer to 100% the relative execution speed gets. Once below cache size 4 the execution time steeply increases. For cache size 1 and a mixed data set the relative execution

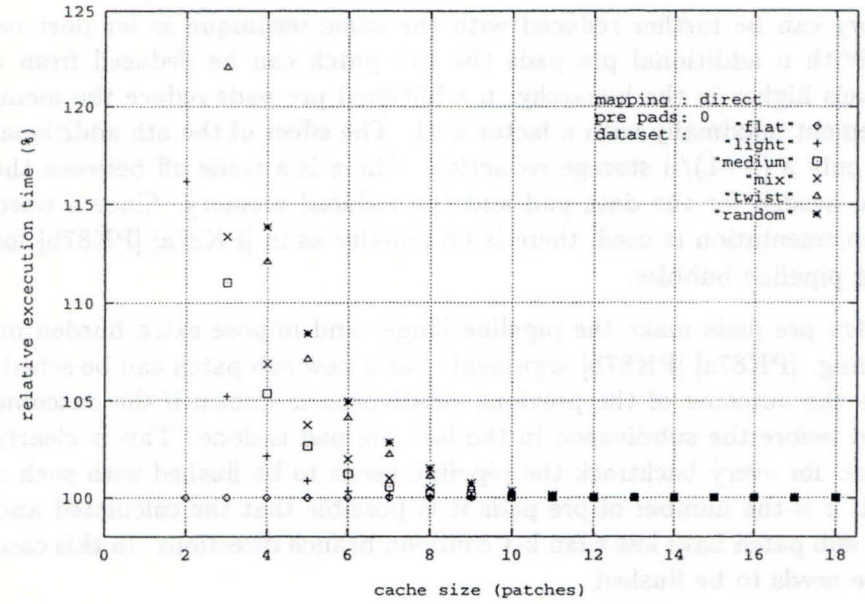

Figure 3: relative execution time vs cache size for different data sets

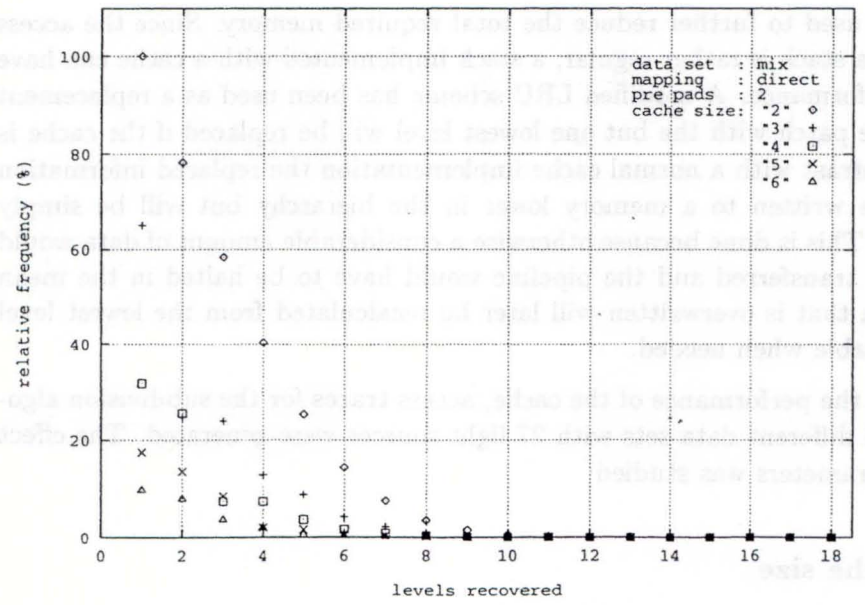

Figure 4: frequency of recovered levels for different cache sizes

time is more than 500%. The flatter the patches in the data set the less influence the cache size has (since there are fewer backtracks).

Fig. 4 shows the relative frequency of the levels that have to be recalculated due to previous overwriting. The bigger the cache the less higher levels need to be recovered. This also means the less recalculation cost.

6.2 Number of pre pads

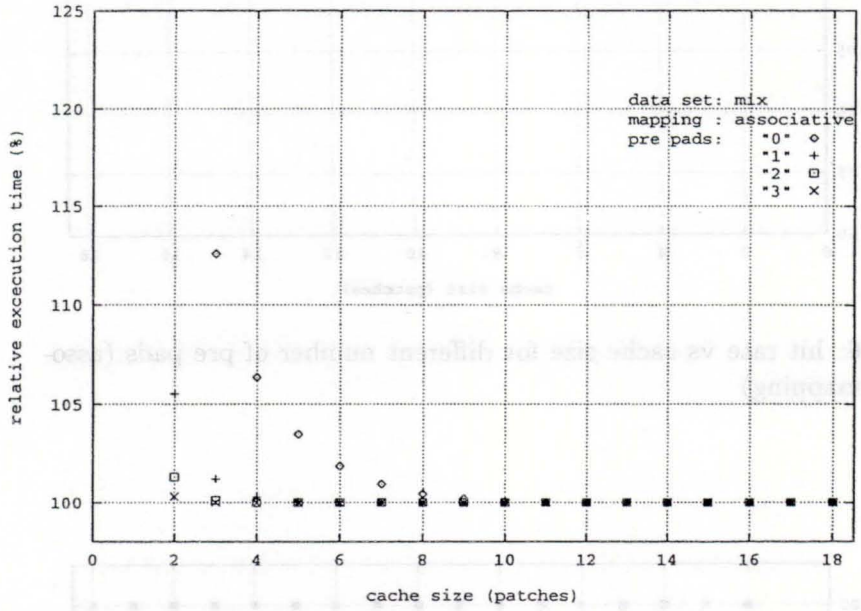

Figure 5: relative execution time vs cache size for different number of pre pads

Fig. 5 shows the influence of cache size on the relative execution speed for a number of pre pad configurations. Increasing the number of pre pads increases the execution speed for a given cache size. The cache size limit below which the execution time steeply increases, decreases with increasing number of pre pads.

As can be seen from fig. 6 the hit rate is almost 100% for cache sizes above 4. For a higher number of pre pads this starts already at cache size 2 and 3.

6.3 Mapping function

A number of different mappings are possible ranging from fully associative over n-way set associative to direct mapped

Fig. 7 shows that there is not much difference for direct and associative mapping. Since direct mapping requires less hardware, direct mapping will be

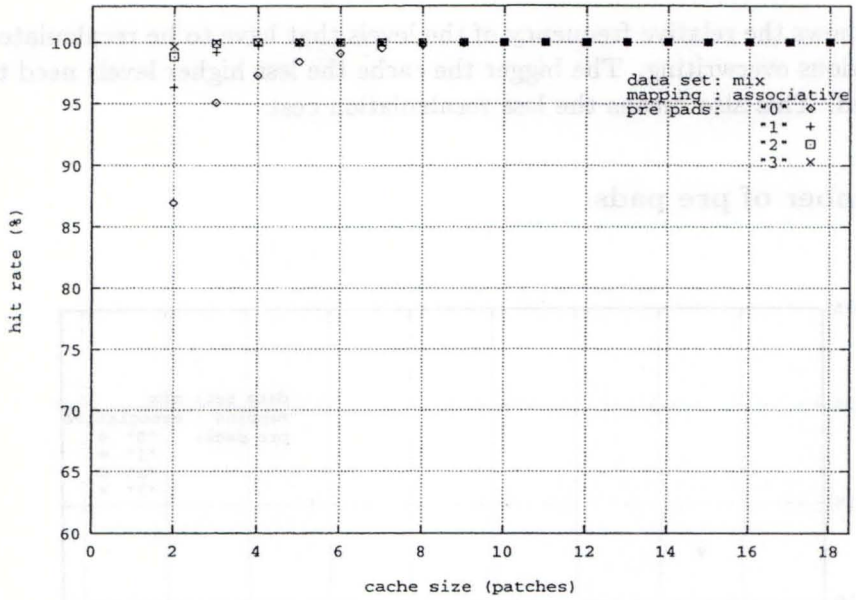

Figure 6: hit rate vs cache size for different number of pre pads (associative mapping)

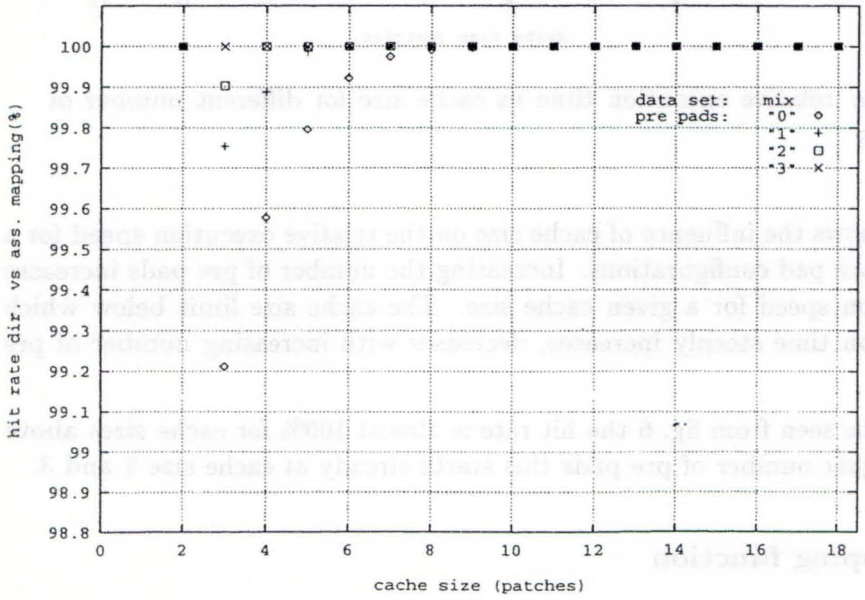

Figure 7: hit rate direct vs associative mapping for different cache sizes and pre pads

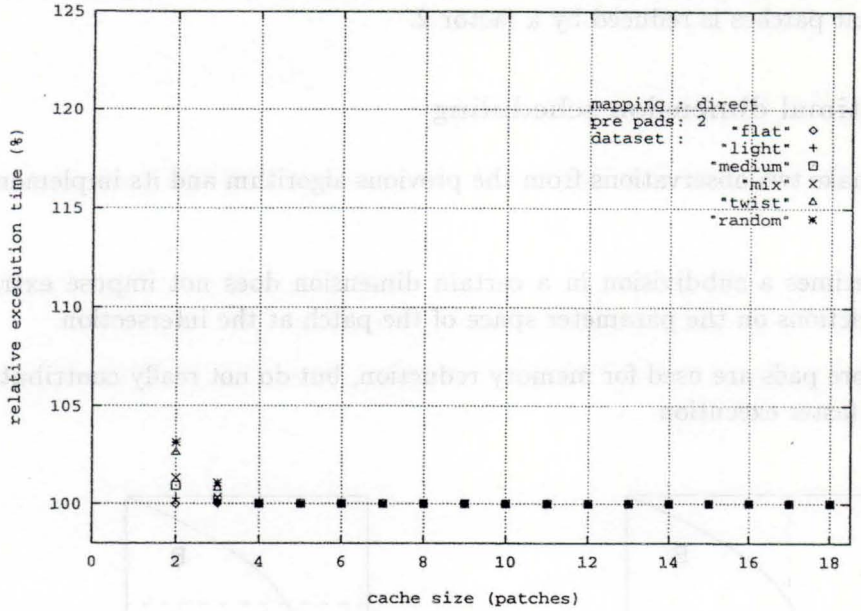

Figure 8: relative execution time vs cache size for different examples with direct mapped cache and 2 pre pads

implemented.

Fig. 8 shows that a cache size of 2 and direct mapping gives a relative execution speed below 103% for all data sets. This configuration will be implemented.

7 High level optimizations

7.1 Independent data scheduling

In a pipelined design it is of high importance that the pipeline can be used efficiently and that there are no pipeline bubbles, stalls or flushes. Since there are data dependencies from one iteration to another, only one iteration per patch can be scheduled at the same time, as has been demonstrated in section 5.3.3.

Due to area limitations, during every iteration only 2 sub patches can be calculated. Two iterations are needed to divide a patch in its 2 dimensional parameter space. A normal implementation would therefore need 3 iterations to calculate all 4 sub patches A_0, A_1, B_0 and B_1 for a patch R: $R \rightarrow (A, B)$, $A \rightarrow (A_0, A_1)$ and $B \rightarrow (B_0, B_1)$. Notice that the 3 operations have a strict ordering.

With small additional hardware, the patch can be subdivided in 4 sub patches in only 2 iterations: $R \rightarrow (A_0, B_0)$ and $R \rightarrow (A_1, B_1)$. This time

the 2 operations have no strict ordering and can be scheduled independently. This means that the total number of patches needed to fill the pipeline with independent patches is reduced by a factor 2.

7.2 Optimal dimension scheduling

One can make two observations from the previous algorithm and its implementation:

- sometimes a subdivision in a certain dimension does not impose extra restrictions on the parameter space of the patch at the intersection
- the pre pads are used for memory reduction, but do not really contribute to a faster execution

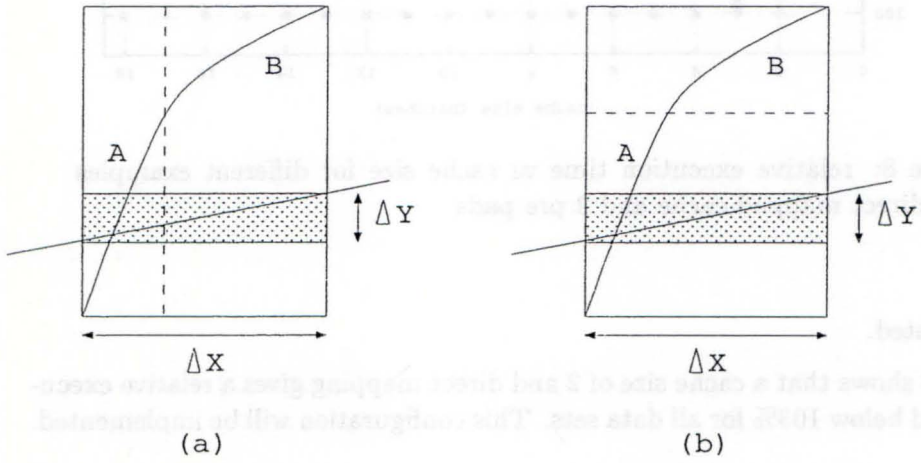

Figure 9: Subdividing a curve in x (a) and in y (b) dimension

Fig. 9 illustrates the first observation for curves: if a curve is subdivided in the x dimension, the bounding boxes for both sub curves are both intersected by the ray and the search space can not be reduced. If the same curve would be subdivided in the y dimension, the search space is effectively reduced. This is possible because the ray traverses the full x dimension of the parent bounding box and only a part of the y dimension.

Within the bounding box the ray covers a box $(\Delta x, \Delta y)$ that has to contain the intersection point (if there is one). If the ray covers a large portion of a certain dimension, it is most unlikely that the ray will only intersect with one sub patch. If the ray covers not more than a certain fraction of the bounding box (here chosen to be 1/2) it is likely that it will only intersect one sub patch.

If in this example, first the control points of the 2 sub curves were to be calculated in the y dimension, the control points of only 1 sub curve are to be calculated in the x dimension. This means that less work has to be performed.

Likewise for patches, only the control points of the sub patches that are not already eliminated by a check in a previous dimension, have to be further subdivided to reduce the search space. Sometimes only one dimension has to be subdivided to reduce the parameter space. The other dimensions need not be further subdivided for the current subdivision level. They will not reduce the search space any further. Since with the pre pads a number of subdivisions can be done at the same time, there is no need to schedule the other dimensions right away: only if the next level is needed, the previous level can be subdivided at the same time in the pre pads. This reduces the number of cycles that is needed to find the intersection.

It is important that a good ordering of the dimensions is obtained. This can be done in the following way. If the following relation is met, it is likely that the subdivision in the x dimension will restrict the search space (ΔRay_i is the extent the ray covers in dimension i, Δt_i is the same extent in parameter space, di the directional coefficient of the ray and ΔBB_i is the extent of the bounding box):

$$\frac{\Delta Ray_x}{\Delta BB_x} < \frac{1}{2}$$

since $\Delta Ray_x = \Delta t \cdot dx$ this becomes

$$2\Delta t \cdot dx < \Delta BB_x$$

This would take extra multiplies and since there is an error on Δt the comparison is imprecise. A better solution is to take the dimension with the smallest ray to bounding box extent ratio:

$$\frac{\Delta Ray_x}{\Delta BB_x} < \frac{\Delta Ray_y}{\Delta BB_y}$$

and thus also:

$$\frac{\Delta BB_x}{\Delta Ray_x} > \frac{\Delta BB_y}{\Delta Ray_y}$$

which since $\Delta Ray_x = \Delta t \cdot dx$ can be reduced to:

$$\frac{\Delta BB_x}{\Delta t \cdot dx} > \frac{\Delta BB_y}{\Delta t \cdot dy}$$

or since $\Delta BB_x/dx = \Delta t_x$ with Δt_x the range of parameter values for intersecting the bounding box planes perpendicular on the x-axis

$$\Delta t_x > \Delta t_y$$

These values have already been calculated for the calculation of the parameter range for the ray for the ray - bounding box intersection. The dimension with the largest Δt range is to be handled first.

The scheduling algorithm for one iteration level then becomes:

- schedule the dimension with largest Δt

- depending on p the number of valid sub patches do:
  - if $(p == 0) \rightarrow$ backtrack
  - if $(0 < p < m) \rightarrow$ goto next iteration level
  - if $(m \leq p \leq 4) \rightarrow$ schedule next best dimension

The parameter m is a compromise between the speed of an iteration and the number of children put on the stack. A good value is 2. In the hardware implementation this will be parameterized.

Some limited performance tests show that this scheduling increases the throughput by 10-50% depending on the data. The flatter the surface, the higher the speed up.

7.3 Sub patch scheduling

If only the closest intersection is to be reported, the sub patches can be sorted in such a way that the first intersection point that is found is the closest one. This requires a *global* sorting of the sub patches. All proposed implementations [PK87a] [PK87b] [BSC89] [Sch87] have however only dealt with a *local* sorting of the sub patches. This means that the first intersection point found is not necessarily the closest one. To guarantee that there is no closer intersection *all* patches on the stack have to examined against the up-to-then closest intersection point. Only [PK87a] [PK87b] handle this correctly. Every subsequently calculated sub patch for which the ray bounding box intersection point is further than the already found one is discarded. Note that if the minimum value of the range Δt were stored for every sub patch on the stack this one iteration per patch on the stack could be saved.

It is however clear that in this case more iterations than necessary are needed to find the closest intersection point since the optimal search path is not followed. An implementation for a global sorting takes a lot of area. Moreover a good upper bound for the stack is difficult to identify and the real upper bound is quite high compared to the normal stack length ($2^n - 1$ vs $(n-2) \cdot 3 + 4$).

The following relations for minimal and maximal bounding box intersection distance for level i and level $j > i$ hold:

$$\begin{aligned} & t_{min,i} < t_{max,i} \\ & t_{min,i+1} > t_{min,i} \quad \Rightarrow \quad t_{min,j} > t_{min,i} \\ & t_{max,i+1} < t_{min,i} \quad \Rightarrow \quad t_{min,j} < t_{min,i} \end{aligned}$$

and thus also for any path from level i to level $j > i$

$$t_{min,i} < t_{min,j} < t_{max,j} < t_{max,i}$$

For two different paths a and b from level i to level j the ordering is kept if and only if

$$t^a_{max,i} < t^b_{min,i}$$

the ordering is then:

$$t^a_{min,i} < t^a_{min,j} < t^a_{max,j} < t^a_{max,i} < t^b_{min,i} < t^b_{min,j} < t^b_{max,j} < t^b_{max,i}$$

and for any level $j > i$

$$t^a_{min,j} < t^b_{min,j}$$

Thus based on a local ordering, a global ordering can be deduced. The only place where global and local ordering are not equal is when the bounding boxes of 2 sub patches overlap. This means that only the sub patches with local overlaps that are still on the stack have to be checked after a first intersection is found.

The hardware implementation will save the local overlap bit for every sub patch that is placed on the stack. After a first intersection is found, only the patches on the stack with the overlap bit set need to be checked. This way the number of iterations *after* the first intersection is found is reduced.

To reduce the number of intersections *before* the first intersection point, it is not only necessary to detect an overlap, but also to know what the actual value is. This needs however excessive register storage ($n^2 \cdot m \cdot 4$ bits for a n bit implementation with m subdivision levels). In this implementation only the 2 smallest values are stored and the rest is discarded. This way a number of overlaps can be correctly handled *before* the first intersection, the rest is handled *after* the first intersection. With a full implementation all of the overlaps would be handled before the first intersection point.

To cope with the bigger stack, only a stack of 1.5 times the depth first stack length was implemented. In most cases the stack will not overflow. If a stack overflow is detected, the control flow will, from that point on, switch over to a depth first search. This way all cases can be correctly handled, and the execution speed is incremented at a small cost.

Some limited performance tests show that this scheduling increases the throughput by 15-40% depending on the data. Especially silhouette edges are handled much faster. The flatter the surface, the higher the speed up.

7.4 Multiple precision

If the normal is to be calculated a bigger number of bits is required to do the intersection calculation. This does not only mean that the data pad gets bigger and slower, but more importantly that also the memory becomes bigger.

A solution is to calculate the intersection point with limited accuracy (and thus also a limited number of bits need to be saved), and once the path is known recalculate the resulting patch with higher accuracy. Intermediate patches don't need to be stored but can be scheduled right after they are produced. Since the path is already known, this can be accomplished with a small number of iterations trough the pre pads (several subdivisions can be done in one iteration). A small performance penalty has thus to be paid since a limited number of extra cycles is needed.

7.5 Projecting

If the patch is projected on a plane perpendicular to the ray, the intersection calculation can be done in $\mathbb{R}^2$ instead of in $\mathbb{R}^3$ [Rog85] [Woo89] [NSK90]. This represents a 33% speed up. Due to a better alignment of the projection of the parameter space of the ray and the real world (the ray coincides with the new z-axis) less sub patches will be valid, and an additional speed up will be obtained. A limited test set show a overall 33-60% improvement. If the intersection point and normal have to be known in real space, the resulting patch can be calculated in $\mathbb{R}^3$. Again this can be done in a limited number of extra cycles. Since only 2 dimensions instead of 3 need to be processed this still represents a net performance gain (20-50%)..

This technique can be elegantly combined with the previous technique for high speed and low storage requirements.

8 Conclusions

It has been demonstrated that a ray-patch intersection algorithm can be efficiently mapped on hardware. A new faster subdivision algorithm has been presented. A numerical analysis shows that the intersection point and normal can be calculated accurately when some provisions have been made. The memory requirement for this recursive algorithm is reduced with pre pads and a cache. Several high level optimizations make efficient scheduling possible and reduce the calculation time compared to previous algorithms.

References

[BSC89] K. Bouatouch, Y. Saouter, and J. C. Candela. A VLSI chip for ray tracing bicubic patches. In W. Hansmann, F. R. A. Hopgood, and W. Strasser, editors, *Eurographics '89*, pages 107–124. North-Holland, September 1989.

[Cla79] J. H. Clark. A fast scan-line algorithm for rendering parametric surfaces. *Computer Graphics*, 13:7–11, August 1979.

[JB86] Kenneth I. Joy and Murthy N. Bhetanabhotla. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 279–285, August 1986.

[Kaj82] James T. Kajiya. Ray tracing parametric patches. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 245–254, July 1982.

[LCWB80] J. Lane, L. Carpenter, T. Whitted, and J. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23(1):23–34, 1980.

[LR80] J. Lane and R. Riesenfeld. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Trans. Pattern Analysis Machine Intell.*, 2(1):35–46, 1980.

[LTM87] Geoff Levner, Paolo Tassinari, and Daniele Marini. A simple, general method for ray tracing bicubic surfaces. In Tosiyasu Kunii, editor, *Computer Graphics 1987 (Proceedings of CG International '87)*, pages 285–302, New York, 1987. Springer Verlag.

[NSK90] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 337–345, August 1990.

[PK87a] R. W. Pulleyblank and J. Kapenga. A VLSI chip for ray tracing bicubic patches. In Wolfgang Strasser, editor, *Advances in Computer Graphics Hardware I, first Eurographics workshop on Graphics Hardware*, pages 125–140, 1987.

[PK87b] Ron Pulleyblank and John Kapenga. The feasibility of a VLSI chip for ray tracing bicubic patches. *IEEE Computer Graphics and Applications*, 7(3):33–44, March 1987.

[Rog85] D. F. Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill, 1985.

[SB86] Michael Sweeney and Richard H. Bartels. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics and Applications*, 6(2):41, February 1986.

[Sch87] Bengt-Olaf Schneider. Ray tracing rational b-spline patches in VLSI. *Advances in Computer Graphics Hardware II, Record of Second Eurographics Workshop on Graphics Hardware*, pages 47–63, 1987.

[Tot85] Daniel L. Toth. On ray tracing parametric surfaces. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 171–179, July 1985.

[vdV89] Alle-Jan van der Veen. Intersection tests for NURBS. In *IEEE Symposium on Computer Architecture and Real Time Graphics*, pages 101–114, Delft, The Netherlands, June 1989.

[Woo89] Charles Woodward. Ray tracing parametric surfaces by subdivision in viewing plane. *Theory and Practice of Geometric Modeling*, pages 273–87, 1989.

[Yan87] Chang-Gui Yang. On speeding up ray tracing of b-spline surfaces. *Computer-Aided Design*, April 1987.