Tracking Data Structures Coherency in Animated Ray Tracing: Kalman and Wiener Filters Approach

Sajid Hussain and Håkan Grahn

Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
{sajid.hussain,hakan.grahn}@bth.se
http://www.bth.se/tek/paarts

Abstract. The generation of natural and photorealistic images in computer graphics, normally make use of a well known method called ray tracing. Ray tracing is being adopted as a primary image rendering method in the research community for the last few years. With the advent of todays high speed processors, the method has received much attention over the last decade. Modern power of GPUs/CPUs and the accelerated data structures are behind the success of ray tracing algorithms. *kd*-tree is one of the most widely used data structures based on surface area heuristics (SAH). The major bottleneck in *kd*-tree construction is the time consumed to find optimum split locations. In this paper, we propose a prediction algorithm for animated ray tracing based on Kalman and Wiener filters. Both the algorithms successfully predict the split locations for the next consecutive frame in the animation sequence. Thus, giving good initial starting points for one dimensional search algorithms to find optimum split locations – in our case parabolic interpolation combined with golden section search. With our technique implemented, we have reduced the "running *kd*-tree construction" time by between 78% and 87% for dynamic scenes with 16.8K and 252K polygons respectively.

1 Introduction

Ray tracing is one of the most widely used algorithms for interactive graphics applications and geometric processing. The performance of these algorithms is accelerated by using bounding volume hierarchies (BVH). BVHs are efficient data structures used for intersection tests or culling in computer graphics. Ray tracing algorithms compute and transverse BVHs in real time to perform intersection test.

While ray tracing has evolved into a real time image synthesis technique in the last decade, more efficient hardware, effective acceleration structures and more advanced transversal algorithms have contributed to the increased performance. Among different acceleration structures [3][4], *kd*-trees have given better or at least comparable performance in terms of speed as compared to others [1]. These structures are more efficient if built using surface area heuristics (SAH) [2].

Interactive ray tracing demands fast construction of BVHs but an optimized fast construction of *kd*-tree is very expensive for large dynamic scenes. Although efforts are being made to optimize *kd*-tree construction for large dynamic scenes [5] [6] [7]

G. Bebis et al. (Eds.): ISVC 2008, Part I, LNCS 5358, pp. 1102–1114, 2008.

[8] [9], there still lies a gulf between *kd*-tree construction and interactive large dynamic scene applications.

In this paper, we present an approach to improve and optimize the construction of *kd*-trees for dynamic ray tracing. We are concerned about the decision of the separation plane location. In most of the dynamic scenes used in research, consecutive frames do not depict considerable differences in terms of geometry information. Our approach is to make use of this particular property for constructing *kd*-tree structures. We start with the approach used in [9] for static scenes, where parabolic interpolation is combined with golden section search to reduce the amount of work done when building the *kd*-trees. We further extend this approach for dynamic scenes and make use of the vector Kalman and Wiener filters to predict the split locations for the next consecutive frame. The same golden section search and parabolic interpolation is then used to find the minimum but this time the predicted location is used as a starting point, hence reducing the number of steps used to find the minimum of the parabolic cost function. The vector Kalman filter results are already presented in another IEEE paper. Here, we extend the work by implementing Wiener filter and comparing the results with that of the Kalman filter. We have evaluated our techniques against a standard SAH algorithm for dynamic scenes with varying complexities and behaviours. With our algorithms, we have achieved average *kd*-tree built times of 30msec for 16-17k triangles scene and 210msec for 252k triangles scene. This corresponds to a reduction of the *kd*-tree construction time by between 78% and 87%.

The rest of the paper is organised as follows. Section 2 gives some related research work on *kd*-tree construction followed by the theory behind SAH based *kd*-trees in section 3. We describe the mathematics behind the Kalman and Wiener filters in section 4 along with our proposed technique in section 5. Section 6 gives our implementation results and some discussion. We conclude the paper in section 7 with future work.

2 Related Work

kd-tree construction has mainly focused on optimized data structure generation for fast ray tracing. The state-of-the-art $O(n \log n)$ algorithm has been analysed in depth by [10] and [11]. Further in [12], the theoretical and practical aspects of ray tracing including *kd*-tree cost function modelling and experimental verifications have been described. Current work in [5] and [13] also aims at fast construction of *kd*-trees. By adaptive sub-sampling they approximate the SAH cost function by a piecewise quadratic function. There are many different other implementations of the *kd*-tree algorithm using SIMD instructions like in [14]. Another approach is used by [15], where the author experiments with stream *kd*-tree construction and explores the benefits of parallelized streaming. Both [5] and [15] demonstrate considerable improvements as compared to conventional SAH based *kd*-tree construction.

The cost function to optimally determine the depth of the subdivision in *kd*-tree construction has been given by several authors. In [16], the authors derive an expression that confirms that the time complexity is less dependent on the number of objects and more on the size of the objects. They calculate the probability that the ray intersects an object as a function of the total area of the subdivision cells that (partly)

contain the object. In [2], the authors use a similar strategy but refine the method to avoid double intersection tests of the same ray with the same object. They determine the probability that a ray intersects at least one leaf cell from the set of leaves within which a particular object resides. They use a cost function to find the optimal cutting planes for a *kd*-tree construction. A similar method was also implemented in [17]. Recently, *kd*-tree acceleration structures for modern graphics hardware have been proposed in [8] and [18], where the authors experiment *kd*-tree for GPU ray tracers and achieve considerable improvement.

3 SAH Based *kd*-Tree Construction

In this section, we give some background about the *kd*-tree algorithm, which will be the foundation for the rest of the paper. Consider a set of points in a space R^d, the *kd*-tree is normally built over these points. In general, *kd*-trees are used as a starting point for optimized initialization of *k*-means clustering [19] and nearest neighbour query problems [20]. In computer graphics, and especially in ray tracing applications, *kd*-trees are applied over a scene S with bounding boxes of scene objects.

The *kd*-tree algorithm subdivides the scene space recursively. For any given leaf node L_{node} of the *kd*-tree, a splitting plane splits the bounding box of the node into two halves, resulting in two bounding boxes, left and right. These are called child nodes and the process is repeated until a certain criterion is met. In [1], the author reports that the adaptability of the *kd*-tree towards the scene complexity can be influenced by choosing the best position of the splitting plane.

The choice of the splitting plane is normally the mid way along a particular coordinate axis [21] and a particular cost function is minimized. In [2], SAH is introduced for the *kd*-tree construction algorithm which works on probabilities and minimizes a certain cost function. The cost function is built by firing an arbitrary ray through the *kd*-tree and applying some assumptions. Fig. 1 uses the conditional probability *P(y|x)* that an arbitrary fired ray hits the region *y* inside region x provided that it has already touched the region *x*. Bayes rule can be used to calculate the conditional probability *P(y|x)* as

$$P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)}. \tag{1}$$

P(x|y) is the conditional probability that the ray hits the region x provided that it has intersected y, and here $P(x|y) = 1$. *P(x)* and *P(y)* can be expressed in terms of areas [1]. In Fig. 2, if we start from the root node or the parent node and assume that *N* is a set of all elements in the root node and the ray passing the root node has to be tested for intersection with all the elements in *N*. If we assume that the computational time it takes to test the ray intersection with elements $n \subseteq N$ is T_n, then the overall computational cost *C* of the root node would be

$$C = \sum_{n=1}^{N} T_n \ . \tag{2}$$

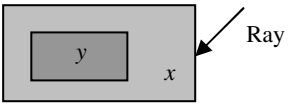

Fig. 1. Visualization of conditional probability *P(y|x)*

After further division of root node (Fig. 2), the ray intersection test cost for each left and right child nodes changes to C_{Left} and C_{Right}. Thus the overall new cost becomes C_{Total} and

$$C_{Total} = C_{Trans} + C_{Right} + C_{Left}, \tag{3}$$

where C_{Trans} is the cost of traversing the parent or root node. The equation can be written as

$$C_{Total} = C_{Trans} + P_{Left} \sum_{i=1}^{N_{Left}} T_i + P_{Right} \sum_{j=1}^{N_{Right}} T_j, \tag{4}$$

where

$$P_{Left} = \frac{A_{Left}}{A} \quad and \quad P_{Right} = \frac{A_{Right}}{A}. \tag{5}$$

Where A is the surface area of the root node and the area of two child nodes are A_{Left} and A_{Right}. P_{Left} and P_{Right} are the probabilities of a ray hitting the left and the right child nodes. N_{Left} and N_{Right} are the number of objects present in the two nodes and T_i and T_j are the computational time for testing ray intersection with the i^{th} and j^{th} objects of the two child nodes. The *kd*-tree algorithm minimizes the cost function C_{total}, and then subdivides the child nodes recursively.

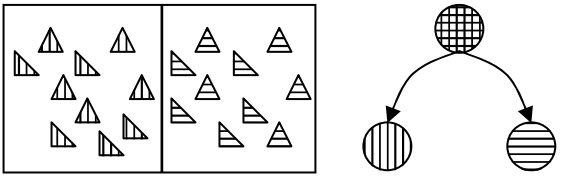

Fig. 2. Scene division and corresponding *kd*-tree nodes

As shown in [15], the cost function is a bounded variation function as it is the difference of two monotonically varying functions C_{Left} and C_{Right}. In [15], this important property of the cost function has been exploited to increase the approximation accuracy of the cost function and only those regions that can contain the minimum have been adaptively sampled. We have used the technique in [9] called golden section search to find out the region that could contain the minimum and combined it with parabolic interpolation to search for the minimum. Further, we predict the minimum of the cost function (split locations) for next consecutive frame using the Kalman filter. We use predicted split locations as starting points for *kd*-tree construction over consecutive frames. In next section, we present some mathematics behind the Kalman and Wiener filter.

4 The Kalman and Wiener Filters

The Kalman filter is named after its inventor Rudolf Emil Kálmán in 1960 [22]. The Kalman filter presents a recursive approach to discrete data linear filtering and prediction problems. The filter estimates the state of underlying discrete time controlled process $x \in \Re^n$ which is presented by the following difference equation.

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}, \tag{6}$$

with the measurement or observation $z \in \Re^m$ and presented by

$$z_k = Hx_k + v_k . \tag{7}$$

The random variables w_k and v_k are process and measurement noises respectively, and assumed to be independent, white and normally distributed with zero mean and covariance matrices Q and R.

$$p(w) \approx N(0,Q) \quad , \quad p(v) \approx N(0,R). \tag{8}$$

Matrix A in equation 7 is an $n \times n$ matrix and it represents the state relationship from previous time step $k-1$ to current time step k. The $n \times 1$ matrix B relates the optional control input u to the state x. The $m \times n$ matrix H in equation 7 relates the state x_k to the measurement z_k. More detailed introduction about the Kalman filter could be found in [24], we will just describe some basic steps of the filter.

The Kalman filter has two main steps called time update (prediction) and measurement update (correction). The prediction state projects the current state estimate ahead in time and the correction state adjusts the projected estimate by an actual measurement at that time. The filter prediction and update steps are described as follows (the details could be found in [23] along with the derivation).

Time update:

$$\begin{aligned} &Prediction: \hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1}, \\ &Error \quad Covariance \quad Projection: P_k^- = AP_{k-1}A^T + Q. \end{aligned} \tag{9}$$

Measurement update:

$$\begin{aligned} &Kalman\,Gain: K_k = P_k H^T \left(HP_k H^T + R\right)^{-1}, \\ &State \quad Ahead \quad Correction: \hat{x}_k = \hat{x}^- + K_k\left(z_k - H\hat{x}^-\right), \\ &Error \quad Covariance \quad Correction: P_k = \left(I - K_k H\right)P_k^-. \end{aligned} \tag{10}$$

In signal processing, the class of linear optimum discrete time filters is collectively known as Wiener filters. The Wiener filter is the filter first proposed by Norbert Wiener in 1949 [25]. Based on statistical approach the goal of the Wiener filters is to filter out noise that has corrupted the signal. Consider $u(t)$ to be a signal input to the Wiener filter, corrupted by additive noise $v(t)$. The estimated output $y(t)$ is calculated by means of a filter $w(t)$ using the following convolution equation

$$y(t) = w(t) * (u(t) + v(t)) \tag{11}$$

We define the error $e(t)$ as

$$e(t) = u(t+\alpha) - y(t), \tag{12}$$

and the squared error is $e^2(t)$. Depending on the value of α the problem can be formulated as prediction ($\alpha > 0$), filtering ($\alpha = 0$) and smoothing ($\alpha < 0$). For discrete time series, consider the block diagram in Fig.3 built around a linear discrete time filter. The filter input consists of a time series and the filter is itself characterized by the impulse response $w(n)$. At some discrete time n, the filter produces output denoted by $y(n)$. This output is used to provide an estimate of desired response denoted by $d(n)$. With the filter input and desired response representing single realizations of respective stochastic processes, the estimation is ordinarily accompanied by an error with statistical characteristics of its own. In particular, the estimation error denoted by $e(n)$ is defined as the difference between the desired response $d(n)$ and the output $y(n)$.

$$e(n) = d(n) - y(n) \tag{13}$$

The requirement is to make the estimation error $e(n)$ as small as possible in some statistical sense. Two assumptions about the filter are made for simplicity. It is linear and operates in discrete time, which makes the mathematical analysis simple and the implementation using digital hardware and software. The impulse response of the filter could be finite or infinite and there are different types of statistical criterion used for the optimization. We use here, Finite Impulse Response (FIR) filter and mean square value of the estimation error. We thus, define the cost function as the mean square error.

$$J = E\left[e(n)e^*(n)\right] = E\left[|e(n)|^2\right]. \tag{14}$$

Where E denotes the statistical expectation operator. The requirement is therefore to determine the operating conditions under which J obtains its minimum value. Further detail reading about the Wiener filter could be found in [26].

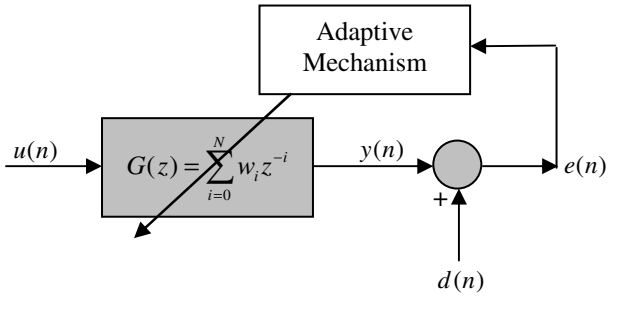

Fig. 3. Block diagram of adaptive control of statistical filtering problem

We have used an adaptive mechanism Least-Means-Square (LMS) to adaptively update the weights of the Wiener filter taps. The order of the Wiener filter increases as we receive more and more samples (samples in this case are the split locations determined for the *kd*-tree construction). We use one Wiener filter with adaptive weights update mechanism for each node in the *kd*-tree of a frame. Fig. 3 also shows the block diagram of adaptive weight control mechanism combined with statistical filtering problem.

The Wiener filter weights are updated by the adaptive mechanism and the following equations govern the adaptive update of the Wiener filter weights.

$$y(n) = \sum_{i=0}^{M-1} w_i(n)u(n-i)\,, \tag{15}$$

$$w_i(n+1) = w_i(n) + \mu u(n-i)e(n) \qquad i = 0,1,....M-1\,. \tag{16}$$

Where $e(n)$ is the error calculated from equation 13 and μ is the step size normally between 0 and 1.

5 Fast Construction of *kd*-Trees

We combine golden section search and parabolic interpolation [9] with the Kalman and Wiener filters to construct *kd*-trees for animated ray tracing. We take advantage of the fact that adjacent frames in animated ray tracing do not depict a dramatic change in most of the animated scenes (we are talking about scenes normally used by the research community in computer graphics).

The algorithm we present here is simple to describe. In our algorithm, we start with the technique described in [9] and construct the *kd*-tree for the first frame of an ani

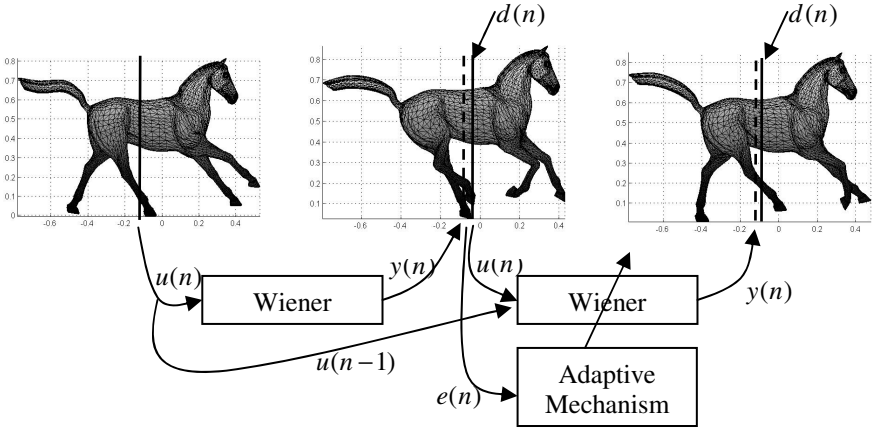

Fig. 4. Wiener filter adaptive prediction

mated scene. Thus, we manage to acquire one sample (sample in our case is the split plane location for a node of the *kd*-tree) for our prediction filters. We then use the Kalman and Wiener filters to predict split plane locations for the next consecutive frame. The algorithm also keeps track of *kd*-tree depth and split orientations. We then apply golden section combined with parabolic interpolation for a one dimensional search of split plane but this time the starting point for the one dimensional search is the predicted split location. This gives us a very fast convergence towards the cost function optimum. Fig. 5 gives an example how the prediction step works for Wiener filters.

The $d(n)$ in Fig.5 is the desired response or in other words is the actual split plane location for a particular axis. We require memory to store the tap values of the Wiener filter $w_i(n)$ and the past inputs $u(n-i)$ in equation 15. The Wiener adaptive filter we have implemented here is an order of 6 (no. of taps) and the adaptive mechanism with $\mu = 0.5$. The beauty of the Kalman filter is that we do not need any previous history to predict the future. This gives us memory free prediction compared to Wiener filter. What we do need is the memory to store the predicted values in the prediction step of the Kalman filter. The same memory locations are updated during the update step. Note that in the Kalman filter step, we use initial split positions information from the first frame and add measurement noise to construct virtual next observations. We then predict the actual split positions for the next frame based on these virtual next observations. In the last step, we update the information for the Kalman filter parameters based on actual split locations information returned by the one dimensional search Algorithm 1. In Wiener filter prediction step, we do not need the virtual information and we predict next split plane position based on previous values. We use the predicted position as a staring point for one dimensional search algorithm (golden section search combined with parabolic interpolation [9]) and refine our prediction results. The refined split plane position is then used as a desired response $d(n)$ and the error $e(n)$ is then calculated for further use in the adaptive weights update mechanism (Fig.5 shows the process visually). The `OptSplit` function in Algorithm 1 takes `KalmanStruct/WienerStruct` which includes all the information about Kalman/Wiener Orientation (`KalmanOrient/WienerOrient`), Kalman/Wiener Depth (`KalmanDepth/WienerDepth`) and Kalman/Wiener predicted optimum split location (`KalmanStartPt/WienerStartPt`). The algorithm itself finds the optimal split orientation for a given depth information and compares it with that of the Kalman/Wiener Orientation. If the two orientations match, the algorithm uses the start point as predicted by the Kalman/Wiener filter. Otherwise, it starts looking for an optimum from extreme positions. Kalman/Wiener Orientation (`KalmanOrient/WienerOrient`), Kalman/Wiener Depth (`KalmanDepth/WienerDepth`) are the two vectors which store the orientation and corresponding depth information from the previous consecutive frame. The objective behind the orientation match is to track the requirements for orientation change because of the dynamic scene. If there is no match, we update the Klaman/Wiener filter parameters with the new orientation and apply the one dimensional search algorithm starting from extreme boundaries of the particular bounding box.

Algorithm 1 – Optimum Split Search

```
function OptSplit(Polygons, AABB, KalmanStruct/WienerStruct)
  Orient = OptSplitOrient(Polygons);
  Extreme = FindExtremes(Polygons, Orient);
  if (Orient = KalmanOrient/WeinOrient and
      Depth = KalmanDepth/WienerDepth)
       Optimum = OptSearch(Polygons, AABB, KalmanOri
                 ent/WienerOrient, KalmanStartPt/WienerStartPt);
  else
       Optimum = OptSearch(Polygons, AABB, Orient);
  return Optimum;
end function
```

6 Results and Discussion

We have tested our algorithm on a variety of animation sequences as shown in Fig. 6. The scenes differ with triangular count and animation behaviour. The scenes consist of regular sized and uniformly distributed triangles. We ran our *kd*-tree construction algorithm and recorded the Kalman and Wiener filters prediction accuracy and the time our algorithm took to build *kd*-tree for each frame in the sequence. We have chosen MATLAB® and C++ for implementation of our algorithm. We have implemented the Kalman and Wiener filters prediction routine in MATLAB® and *kd*-tree construction routine in C++. The *kd*-tree construction is linked in MATLAB® through Dynamic Link Library (DLL). Routines for PLY file reading are also implemented in MATLAB®. The timing results shown in this paper are only for *kd*-tree construction in DLL. We have performed all the simulations on a workstation with an Intel Core2 CPU, 2.16 GHz processor and 2GB of RAM.

The scenes we have used in our simulations vary in terms of their complexities and behaviors. Fig. 6 shows three different animation sequences. In Fig. 7 we analyze cost functions change in cloth-ball (92.2K – 73 Frames) animations for the two axis (y and z) as shown in Fig. 6 (cloth-ball animation) for each scene and for only root node split positions. We also plot actual split positions and predicted split positions of the Kalman and Wiener filters. Note that the actual split positions have been calculated on bases of Surface Area Heuristics (SAH).

Let's analyse Fig. 7 closely, the upper two sub-figures in Fig.7. (left to right) show cost function shift for each frame in cloth-ball animation sequence for y and z axis respectively. The minimums of these parabolic cost functions are the optimum split plan locations. The bottom sub-figure in Fig. 7 shows the actual optimum split plan locations change over time and predicted split plan locations by the Kalman and Wiener filters (time axis is no. of frames in this case) for only y axis of cloth-ball animation. Note the settling time for Wiener filter in this case. The adaptive mechanism controls the tap weights of the Wiener filter and tries to minimize the mean square error between desired and predicted response. Our algorithm has successfully predicted the split plan locations for consecutive frames. Hence, provides good initial guess for one dimensional search (parabolic interpolation combined with golden section search). If we closely analyse the prediction curves, we see that in almost all the cases, the prediction error is very small. Since, the entire scene dataset exhibits a strong coherency between consecutive frames; we have successfully exploited this property here.

Table 1. Conventional vs Modified *kd*-Tree Build Time

Scene	Primitives	Conventional *kd*-Tree Build (msec)	Modified *kd*-Tree Build (msec)			Time Reduced Kalman/ Wiener
			Initial Build	Running Build		
				Kalman	Wiener	
Horse Animation	16.8K	135	135	30	28	78% / 79%
Elephant	84.6K	710	710	105	95	85% / 86%
Cloth-Ball	92.2K	802	802	120	105	85% / 87%
Bunny Dragon	252.5K	1610	1610	210	195	87% / 88%

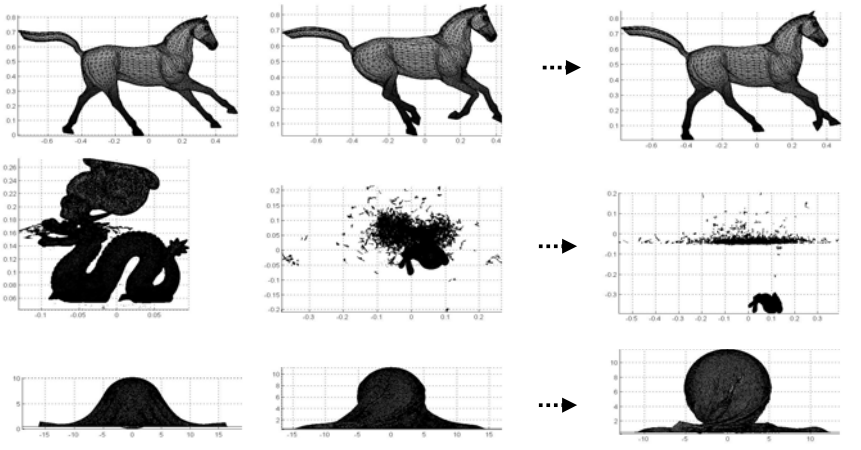

Fig. 6. Animation sequences (top to bottom): Horse (16.8K - 48 Frames), BunnyDragon (252.5K - 16 Frames) and ClothBall (92.2K - 73 Frames)

Although, in all these scenes, there remain a constant number of polygons (triangles) throughout a particular animation sequence, we see a random behaviour in the *kd*-tree build time especially for Kalman filter. The phenomenon occurs due to random noise added by the Kalman filter prediction steps, where we have constructed the virtual observations by adding the measurement noise from equation 8. The added noise maximum error difference is not greater than 1msec.

Table 1 shows the time difference between an initial build and a running build of *kd*-tree data structures for each animation sequence used in this paper. See the considerable improvement in the build time in running mode for both Kalman and Wiener filters. We have not yet added the overhead of the Kalman and Wiener filters in Table 1. In MATLAB®, the Kalman filter's average aggregated overhead is approx. 400-450 msec and that of Wiener filter is approx. 500-550 msec for the whole sequence of 50 frames with average of 84K polygons (triangles) in each frame. We expect this time down to 100-150 msec for Kalman filter and 150-200 msec for Wiener filter if efficiently implemented in C++. So, in worst case we could add 4-5 msec per frame.

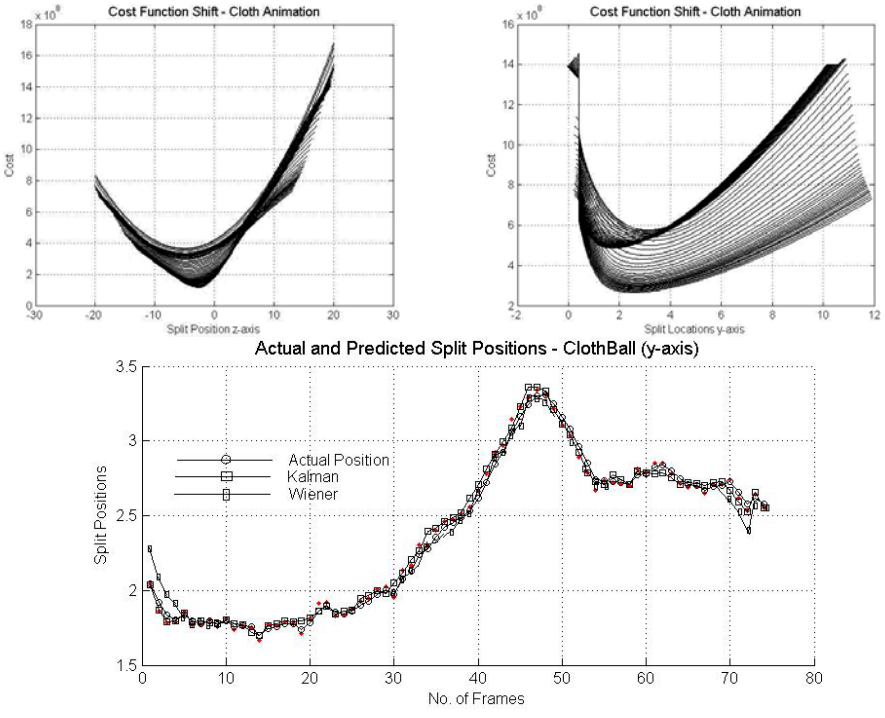

Fig. 7. Actual and predicted split positions for Kalman and Wiener filters

7 Conclusion and Future Work

We have presented an algorithmic speedup technique for fast *kd*-tree construction for animated ray tracing. The optimum split location search for *kd*-tree construction is the main time consuming job. As many of the animation sequences used by research community for animated ray tracing exhibit strong data structures coherency properties, we have made use of a Kalman and Wiener filters for predicting the next possible data structure (*kd*-tree in our case) state of the animated sequence.

We use here the Kalman and Wiener filters and load it with initial split plan locations (we build *kd*-tree for starting frame in the sequence based on the technique described in [9]). The filters then predict the next possible split locations for the next frame in the sequence. We use these predicted locations as starting points for the one dimensional optimum search algorithm. With best initial guess, the algorithm exhibits very fast convergence and we see the results quite promising for the running *kd*-tree build time as compared to static or initial *kd*-tree build. In the case of Kalman filter we achieve 78% to 87% increase in *kd*-tree construction time for the scenes with as low as 17K and as high as 252K polygons. The increase in *kd*-tree construction time for Wiener filter is 79% to 88% for scenes with same complexities.

We have implemented our proposed model in MATLAB® and C++. Main prediction engine of the Kalman and Wiener filters is implemented in MATLAB®. C++ handles the *kd*-tree construction routines. We have demonstrated a considerable decrease in build time as compared to standard SAH based *kd*-tree.

References

[1] Havran, V.: Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague (2001)

[2] MacDonald, J.D., Booth, K.S.: Heuristics for Ray Tracing Using Space Subdivision. In: Graphics Interface Proceedings 1989, Wellesley, MA, USA, June 1989, pp. 152–163. A.K. Peters, Ltd. (1989)

[3] Stoll, G.: Part I: Introduction to Realtime Ray Tracing. In: SIGGRAPH 2005 Course on Interactive Ray Tracing (2005)

[4] Zara, J.: Speeding Up Ray Tracing - SW and HW Approaches. In: Proceedings of 11th Spring Conference on Computer Graphics (SSCG 1995), Bratislava, Slovakia, pp. 1–16 (May 1995)

[5] Hunt, W., Stoll, G., Mark, W.: Fast kd-tree Construction With An Adaptive Error-Bounded Heuristic. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 81–88 (September 2006)

[6] Wald, I., Havran, V.: On Building Fast kd-trees For Ray Tracing, and on Doing That In O(N log N). In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 61–69 (September 2006)

[7] Woop, S., Marmitt, G., Slusallek, P.: B-kd trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In: Proceedings of Graphics Hardware (2006)

[8] Foley, T., Sugerman, J.: kd-tree Acceleration Structures For A GPU Raytracer. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 15–22 (2005)

[9] Hussain, S., Grahn, H.: Fast kd-Tree Construction for 3D-Rendering Algorithms like Ray Tracing. In: Bebis, G., Boyle, R., Parvin, B., Koracin, D., Paragios, N., Tanveer, S.-M., Ju, T., Liu, Z., Coquillart, S., Cruz-Neira, C., Müller, T., Malzbender, T. (eds.) ISVC 2007, Part II. LNCS, vol. 4842, pp. 681–690. Springer, Heidelberg (2007)

[10] Wald, I.: Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, Saarbrucken, Germany (2004)

[11] Havran, V.: Heuristic Ray Shooting Algorithm. PhD thesis, Czech Technical University, Prague (2001)

[12] Chang, A.Y.: Theoretical and Experimental Aspects of Ray Shooting. PhD Thesis, Polytechnic University, New York (May 2004)

[13] Havran, V., Herzog, R., Seidel, H.-P.: On Fast Construction of Spatial Hierarchies for Ray Tracing. In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, pp. 71–80 (September 2006)

[14] Benthin, C.: Realtime Raytracing on Current CPU Architectures. PhD thesis, Saarland University (2006)

[15] Popov, S., Gunther, J., Seidel, H.-P., Slusallek, P.: Experiences with Streaming Construction of SAH KD-Trees. In: Proceedings of IEEE Symposium on Interactive Ray Tracing, pp. 89–94 (September 2006)

[16] Cleary, J.G., Wyvill, G.: Analysis Of An Algorithm For Fast Ray Tracing Using Uniform Space Subdivision. The Visual Computer (4), 65–83 (1988)

[17] Whang, K.-Y., Song, J.-W., Chang, J.-W., Kim, J.-Y., Cho, W.-S., Park, C.-M., Song, I.-Y.: An Adaptive Octree for Effi¬cient Ray Tracing. IEEE Transactions on Visualization and Computer Graphics 1(4), 343–349 (1995)
[18] Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive kd-tree GPU Raytracing. In: Symposium on Interactive 3D Graphics. I3D, pp. 167–174 (2007)
[19] Redmonds, S.J., Heneghan, C.: A Method for Initializing the K-Means Clustering Algorithm Using kd-trees. Pattern Recognition Letters 28(8), 965–973 (2007)
[20] Stern, H.: Nearest Neighbor Matching Using kd-Trees. PhD thesis, Dalhousie University, Halifax, Nova Scotia (August 2002)
[21] Kaplan, M.: The Use of Spatial Coherence in Ray Tracing. In: ACM SIGGRAPH 1985 Course Notes, vol. 11, pp. 22–26 (July 1985)
[22] Kalman, R.E.: A New Approach to Linear Filtering and Prediction Problems. Transaction of the ASME—Journal of Basic Engineering, 35–45 (March 1960)
[23] Welch, G., Bishop, G.: An Introduction to Kalman Filter. Department of Computer Science, University of North Carolina (July 2006)
[24] Grewal, M.S., Andrews, A.P.: Kalman Filtering, Theory and Practice. Prentice Hall, Englewood Cliffs (1993)
[25] Wiener, N.: Extrapolation, Interpolation, and Smoothing of Stationary Time Series. Wiley, New York (1949)
[26] Haykin, S.: Adaptive Filter Theory, 3rd edn. Prentice Hall, New Jersey (1996)