An Error-Resilient Arithmetic Coding Algorithm for Compressed Meshes

Zhi-Quan Cheng, Bao Li, Kai Xu, Yan-Zhen Wang, Gang Dang, Shi-Yao Jin
PDL Lab., NUDT University, P.R. China
Cheng.zhiquan@gmail.com

Abstract

The effort of on-the-fly accessing 3D contents over the internet has been done in recent years. And 3D streaming has been investigated to represent 3D models in the compact format and progressively transmit them on the limited-bandwidth and lossy channel. In the paper, an error resilient 3D mesh coding algorithm is presented, which employs an extended multiple quantization (EMQ) arithmetic coder method, inspired by the error-resilient JPEG 2000 image coding standards. With periodic arithmetic coder restarting and termination markers, the error resilient EMQ coder divides bit stream into little independent parts and enables basic transmission error containment. Furthermore, the EMQ coder has the intrinsic capacity of handling noise by using the maximum a posteriori (MAP) decoder. Experiments show that the method improves the mesh transmission quality in a simulated network environment.

1. Introduction

Now, the technologies for transmission of 3D models face an increased demand in professional and private scenarios. 3D mesh compression [1] can be employed to reduce the file size and end-to-end delay in the compact single-resolution [2] or progressive representation [3]. Progressive transmission is scalable with respect to both network bandwidth and the user's quality measurement. However, most of the type algorithms have assumed an error-free transmission channel, or storage medium for that matter. This is a fairly un-reasonable assumption when the network layer has weak error protection. If the network layer does not provide strong error protection, the decoded model can be severely affected. In fact, the real internet is lossy based on UDP/IP protocol, and has limited bandwidth. To meet this gap, error control schemes must be applied.

Arithmetic coding, which has been widely used in mesh compression, is very sensitive to bit errors. An error bit will, at some point, affect the interval test, thus an incorrect symbol will be decoded. All sequent symbols will also be erroneously decoded as the arithmetic coder gets out of sync. Even if the arithmetic coder was removed, the coefficient bit modeling, which generates the coded symbols, is also sensible to errors and can get out of sync after a bit error. A single bit error will thus alter all the following coded data to useless. For instance, if the error occurs in the middle of the bit stream, half of the model is lost. Even worse, if the error is not detected, arbitrary data is decoded and a model that has no resemblance to the original is decoded. Missing packets have a similar effect. In the following sections we will present methods that allow limiting the extent up to which bit errors affect the model data. These methods are inspired by the error resilience mechanisms in JPEG 2000 [4] and similar image or video coding standards, which rely on data partitioning. A subset of the extensions also enables other functionality such as random access and on-the-fly reordering, as we later explain. We should note that the proposed extensions are intended only for modest bit error rates (BER), such as the residual errors left by network or channel level forward error correction (FEC).

In the paper, error detection is achieved through checks in the residual redundancy on the coded data and insertion of segment markers. The mechanism is shown to provide effective means to detect and contain transmission errors. It should be noted that the bitstream header contains very sensible information: number of surface, bounding box size and offset, codec options, predictor coefficients and quantized step sizes. It is therefore of paramount importance that its transmission be error free. As the size of the header is very small, it is feasible to apply strong error correcting codes and automatic repeat request (ARQ) technique to guarantee proper decoding with only minimal cost. Hereafter, it is assumed that the header is
transmitted without any error.

The paper is organized as follow. Section 2 summarizes the related error-resilient transmission methods. Section 3 describes our encoding algorithm, which consists of data partitioning and basic error detection. In Section 4, we evaluate the error resilience of the proposed algorithm. Finally, we conclude this paper in section 5.

2. Related works

Since Bajaj et al. [5] first proposed an error-resilient streaming method for compressed VRML models, many algorithms have been developed to robustly transmit 3D meshes. The error control schemes can be broadly classified into three categories: sender-based, network-based, and receiver-based.

Sender-based error control. The sender estimates the channel resources (loss ratio) and protects the bit-stream by adding redundant bits to be able to recover lost data on the client side. The method proposed by Bajaj et al. [5] is a typical sender-based error control way, where the encoded bit-stream is divided into layers according to the depth-first order of the vertices. As a result, each layer is independent of all other parts of the model and losing any layer will not affect the decoding of other layers. Subsequently, Yan et al. [6][7] finish their error resilient coding of 3D models by using the mesh segmentation scheme to prevent channel error propagation. While the series of algorithms, introduced by AI-Regib et al.[8][9], are scalable to both channel bandwidth and channel error characteristics. The bit-budget allocation method assigns an optimal FEC bit budgets, and distributes the error-protection bits among the transmitted layers to maximize the expected decoded model quality. Their experiments showed that the quality of the decoded model degrades more gracefully as the packet-loss rate increase. Since sender-based methods are highly effective [10] and they require little processing on the client side, our algorithm is among them.

Network-based error control. Reliable transport TCP protocol has been the most commonly used on the Internet, and it is an example of this approach. However, TCP is not applicable for the time-sensitive applications due to its retransmission and congestion control mechanisms. For example, considering the case when a client requests a 3D model to be downloaded within a limited time, if the channel suffers from packet losses, then many TCP packets will be lost. Thus, unfortunately, it increases the download time. On the other hand, UDP are streaming friendly but not reliable. Therefore, AI-Regib et al. [11] proposed 3TP protocol, which is a hybrid protocol to take advantage of both TCP and UDP. They showed that by intelligently selecting certain parts of the 3D data transmitted via TCP and others via UDP, the minimum distortion and delay should be obtained. Moreover, they showed that the structural data affects distortion of the reconstructed mesh more than the geometric data does. Li and Prabhakaran [12] proposed a generic 3D middleware between the 3D application layer and the transport layer for the transmission of triangle based progressively compressed 3D models. The middleware could consider end user hardware capabilities to effectively save the data size for network delivery, and introduced a minimum cost dynamic reliable set selectors to choose the transport protocol for each sub-layer depending on the relative importance and real-time network traffic.

Receiver-based error control. The visual quality of corrupted 3D meshes can be improved by concealing or interpolating lost regions. In 3D data processing, error concealment can be considered as hole-filling [13] and surface reconstruction [14] problem. Such methods exploits the smoothness properties of the model in recovering lost parts, and they generally do not add any redundant bits. Usually, receiver-based algorithms work independently of the channel behavior and they require fairly high computational power at the client to perform processing in a timely fashion. Bischoff and Kobbelt [15] exploited the geometry coherence of the received data. The model at the sender was re-sampled to ensure such coherence between samples, and on the client side, the received samples were used to construct an approximation of the original 3D model.

3. Error Resilient Coding Algorithm

Our objective is to design an error-resilient transmission method that is respect to channel error. In the paper, we utilize our progressive degree-driven algorithm [16] to compress the 3D models. The improved method iteratively applies the valence-driven decimating conquest and the cleaning conquest in pair to achieve connectivity compression. And in the geometrical compression, we use the barycenter prediction and the local approximate Frenet coordinate frame to separate normal and tangential components to further reduce the bit rates. Especially, the quantization in tangential components is less than the normal, since they has less contribution to the reconstructed model.

3.1. Standard MQ Coding
MQ coder [4] keeps the two features of the Q-coder in IBM’s Adaptive Bilevel Image Compression(ABIC): multiplication-free approximation and bit stuffing, and integrates conditional exchange and the Bayesian learning of probability-estimation state machine. The input of MQ coder is D and CX. D = \{d_0, \ldots, d_L\}, corresponds to a binary code, and CX = \{x_0, \ldots, x_L\} is the context of D. MQ coder recursively partitions the probability interval: at each time when the coder receives D, the interval is split into two subintervals(Fig. 1. a). In MQ coders, the decision bits D is not coded by 0 and 1, but classified into MPS(Most probable symbol) and LPS(Least probable symbol). If \(Q_e\) is the probability of LPS, then the probability of MPS, referred as \(P_e\), equals to \(1 - Q_e\). The arithmetic coder partitions the current interval into most-probable-interval and least-probable-interval. The symbols A and C are used to denote the current interval, corresponding to the lower limit and A the length of current interval, respectively. Thus the most-probable-interval can be expressed as \([C + AQ_e, C + A]\), and \([C, C + AQ_e]\) for the least-probable-interval. To avoid multiplications, the length of the interval, A, is approximated by \(A^* = 1 - Q_e\), so \(A_i, i=0, \ldots, L\) can be approximated by \(Q_e\). Then now the most-probable-interval can be denoted by \([C + AQ_{e_i}, C + A]\), and \([C, C + AQ_{e_i}]\) for the least-probable-interval. The following equations can be induced:

**MPS coding:**

\[
\begin{align*}
C_{i+1} &= C_i + Q_e \\
A_{i+1} &= A_i - Q_e
\end{align*}
\]

**LPS coding:**

\[
\begin{align*}
C_{i+1} &= C_i \\
A_{i+1} &= Q_e
\end{align*}
\]

If the data to be coded is MPS, then the most-probable-interval is selected to be the current interval of next iteration coding. Similarly, if the data to be coded is LPS, then the least-probable-interval is selected to be the current interval of next iteration coding. Although arbitrary value of current interval can be used to denote coding data, C should be outputted after coding and delivered as the result of compression. In the same way, the arithmetic decoder can determine whether the inputted bit-stream belongs to most-probable-interval or least-probable-interval, based on the bit-stream itself and CX, then carries decompression on the data.

### 3.2. Extended MQ coder with Data Partitioning

The major problem with respect to error-prone transmission is that a model is coded as a single unit and a single bit error makes all the subsequent data unusable. A progressively compressed mesh has inherent data partitioning, as different multi-resolution levels can be independently coded. Even if the multi-resolution surfaces were handled independently by the prediction and coefficient bit modeling stages, they are coded in the same arithmetic coder bit-stream. This is in general beneficial, in particular to models with small surfaces, since the adaptive arithmetic coder processes enough data to accurately learn its statistics. For error resilience this is however catastrophic. The arithmetic coder needs therefore to be periodically terminated and restarted. The overall bit-stream will be hence made of independent arithmetic coder bit-streams that are concatenated together. We choose to restart the arithmetic coder every level, and exploit two special features of the MQ arithmetic coder.

On the one hand, MQ coders can be extended by the introduction of a forbidden region with probability \(Q_f\). The modified MQ is referred as EMQ(extended multiply quantization). As shown in Fig. 1. b, the probability interval is partitioned into \(Q_f\), LPS and MPS, and the lower limit C is adjusted. Obviously, at each iteration \(Q_f\) is added to C, in order to skip the forbidden region. The encoder complexity is almost the same as standard MQ, since only one supplementary summation is needed. It is more important that the forbidden region allows for error detection at the decoder side, which is able to stop decoding as soon as the received code \(C_f\) falls into the forbidden region, \(C_f < Q_f\). Apparently, the error detection probability increases as the number of decoded symbols increases; error detection is better when \(Q_f\) is larger; and when \(Q_f = 0\), EMQ degenerates to MQ. Chou and Ramchandran [17] have proved that the coding redundancy, conveyed by \(Q_f\), amounts to \(-log_2(1-Q_f)\).

![Figure 1. Probability intervals in MQ and EMQ coder.](image)

On the other hand, we exploit the termination makers of the MQ coder. The MQ coder never generates 2-byte words in the range 0xFF90 to 0xFFFF, in hexadecimal. These sequences are denoted
termination marker makers. When the decoder encounters a
termination marker it locks into a mode where it
behaves as if the bit-stream continued with an infinite
sequence of repeatedly 1 bits. In this mode no more
bit-stream data is consumed. Each time the MQ coder
is to be restarted flush its internal state and include a
termination marker, just before starting. Since the
termination markers can not be emulated by coded data,
unless a bit error synthesizes one, a decoder can
identify the individual bit-stream segments generated
by the MQ coder. Since there are a fair number of
different termination markers we successively use the
ones in the 0xFF90 to 0xFFCF range to signal a
sequence index with period 64(every 100 bytes, for
example). When the last surface is coded we use a
special marker, 0xFF00, to signal the end of bit-stream,
which is depicted in Fig. 2. Note that a termination
marker also included right after the bit-stream header
and before the first surface. This allows to detecting
packet lossless at the start of the bit-stream. The
average cost of including a marker is 3.5 bytes, since
two bytes are required for the marker and 1.5 bytes are
consumed when flushing the internal MQ coder state.

![Figure 2. Bit-stream with periodic
termination markers.](image)

The decoder periodically checks the received code.
If the specific termination markers can’t be found at
the destined positions, errors in the transmission will
be figured out.

Whenever an error resilient decoder detects an error
it will locate the next bit-stream segment by searching
for the next termination marker. It can also use the
sequence index and end of bit-stream markers to detect
packet loss and/or makers that have been affected by
bit errors.

To gain basis error detection, we can combine the
two methods aforementioned together. More
specifically, we can further partition the bit stream by
inserting periodic termination markers in EMQ. What
is more important, that this EMQ coder can be applied
into any existing 3D geometric compression algorithm,
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
we replace the adaptive arithmetic coder [18] in [16],
and the incurred overhead is very small. For instance,
decoding step. For example, at the $j$-th level of the decoding tree, the two departing transitions are labeled with the APPs $c_j' = 0$ and $c_j' = 1$, respectively. Thus the decoder can obtain decoded bit $d_j$, according to the context $x_j$. If an error is detected in a branch, the branch will be pruned. Otherwise, the decoder will turn into the next state, $\sigma_{j+1}$. It is clearly that the goal of decoder is to pick up the path with maximum APP.

We test the MAP decoding algorithm in a wireless 3G network. Take the david model in [16] as an example, whose connectivity compression rate and geometric compression rate is 5.9bpv and 15.6bpv, respectively. The model is transmitted in base station controller (BSC) channel and additive white Gaussian noise (AWGN) channel, with two different bit error rates (BER) $10^{-3}$ and $10^{-4}$. Table 2 is the error correction results of EMQ, under different configurations. $Q_f$ can be 0.005, 0.02, 0.05 and 0.08. The length of interval separated by termination marks is 100 bytes. The results is estimated by peak signal to noise ratio (PSNR), computed as linear average of the mean square error (MSE) over 100 independent simulations.

<table>
<thead>
<tr>
<th>$Q_f$</th>
<th>BER $= 10^{-3}$</th>
<th>BER $= 10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BSC AWGN</td>
<td>BSC AWGN</td>
</tr>
<tr>
<td>0.005</td>
<td>27.7 36.5</td>
<td>19.4 34.5</td>
</tr>
<tr>
<td>0.02</td>
<td>35.4 36.1</td>
<td>26.6 35.5</td>
</tr>
<tr>
<td>0.05</td>
<td>34.9 35.2</td>
<td>31.3 35.1</td>
</tr>
<tr>
<td>0.08</td>
<td>34.4 34.4</td>
<td>33.1 34.3</td>
</tr>
</tbody>
</table>

Our very simple error detection algorithm proves to be surprisingly effective, in particular for meshes with uncomplicated surface, as the most severe errors are detected. Examples of decoded models are shown in Fig. 4 for BER of $10^{-4}$. The decoder simply drops a surface when it detects an error. We can see that the severely distorted surfaces are in general avoided by this simple error detection mechanism. Occasionally, some severe errors go undetected and result in highly distorted surfaces. Nevertheless, we should point that since the length of the bit-stream is small for some models, very often no errors will hit the bit-stream and they will be perfectly decoded.

4. Conclusion

We presented a novel algorithm for error resilient coding of 3D meshes, after reviewing error control schemes of robust 3D model transmission over network. The algorithm makes use of an extended multiple quantization (EMQ) arithmetic coder method, inspired by the error-resilience mechanisms presented in JPEG 2000 image coding standards. The encoder uses a shape adaptive mesh segmentation scheme to alleviate the effect of error propagation. An input mesh surface is coarsely divided into parts, and each part is further progressively compressed by the EMQ coder with periodic arithmetic restarting and terminating.
markers to enable transmission error containment.

We believe that much room remained for the further improvement. Simple heuristics should be added to assess the required amount of splitting, given the channel bit error rate and the application tolerable surface loss probability. Error concealment strategies more elaborate than simple surface discarding should also be proposed. Finally, the application of termination markers to bit-stream reordering for optimal view-dependent transmission and random access of large models can be investigated.

Acknowledgements

The work is funded by National Natural Science Foundation of China under Grant No. 60773020.

References