Preprint — Artical in Vision, Modeling, and Visualization (2011)
Peter Eisert, Konrad Polthier, and Joachim Hornegger (Eds.)

Adaptive Level-of-Precision for GPU-Rendering

Quirin Meyer[1] Gerd Sussner[2] Günter Greiner[1] Marc Stamminger[1]

[1]Computer Graphics Group, University Erlangen-Nuremberg, [2]RTT AG

Abstract

Video memory is a valuable resource that has grown much slower than the rendering power of GPUs over the last years. Today, video memory is often the limiting factor in interactive high-quality rendering applications. The most often used solution to reduce memory consumption is to apply level-of-detail (LOD) *methods: only a simplified version of the mesh with less vertices and triangles is kept in memory. In this paper we examine a simple orthogonal compression approach that is mostly neglected: adapting the* level-of-precision (LOP) *of vertex data. The main idea is to quantize vertex positions according to the current view distance, and adapt precision by adding or removing single bit planes. We provide an analysis of the resulting image error, and show that visual artifacts can be avoided by simply constraining the quantization for* critical *vertices. Our approach allows both random access on vertex data as well as quickly switching between LOP. In experiments we found that our approach compresses vertex positions by about* 70 % *on average without loss in rendering performance or image quality.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Today's GPUs process more than a billion triangles per second, so scenes with enormous geometric detail can be rendered interactively. This is particularly useful in high-quality rendering applications such as point-of-sale presentations or product design reviews, where highly detailed construction data is to be rendered faithfully. Including variants and animations, a scene can easily contain several million vertices. Also in computer games, where assets are much more optimized for a low vertex count, geometry becomes more and more complex to increase realism. Adding normal and tangent vectors, texture coordinates and maps, and index buffers, GPU memory quickly becomes the limiting factor. There has been intensive research on texture compression, but much less on the compression of geometry data for GPUs. The most efficient geometry compression techniques utilize sequential decompression, which is typically too slow for real-time rendering. An alternative are *level-of-detail (LOD)* methods, where a simplified version of a mesh with a reduced number of vertices and triangles is rendered. Typically, the simplification level is adapted to the current view distance by adding or removing vertices. This happens continuously, or by swapping a discrete set of simplifications. These approaches save video memory if only a single simplified version of the mesh is present on the GPU.

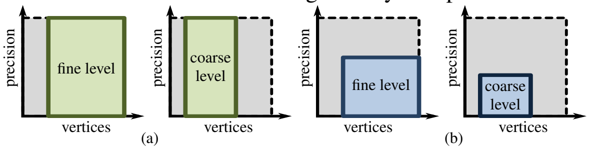


Figure 1: (a) LOD methods vary the number of vertices. (b) Simultaneously, our *level-of-precision (LOP)* methods vary the numerical accuracy.

In this paper we focus on another, simple possibility to save video memory, which is largely ignored, but has high potential: *level-of-precision (LOP)*. Conventionally, numerical precision of vertex positions is constant. However, depending on the viewing distance, lower precision, i.e. a smaller number of bits per vertex, is acceptable, as the quantized vertices are projected into the same sub-pixels. This idea is depicted in Figure 1: An LOD method (a) only stocks a subset of all vertices in video memory. If the LOD is adapted, vertices are removed and added, so the working set in video memory increases or shrinks. However, we can adapt the precision of vertex data simultaneously (b). For a coarse LOD, a low numerical precision is acceptable. In this paper, we consider LOP for vertex positions only, but other vertex attributes such as shading normals or texture coordinates can be compressed in a similar way.

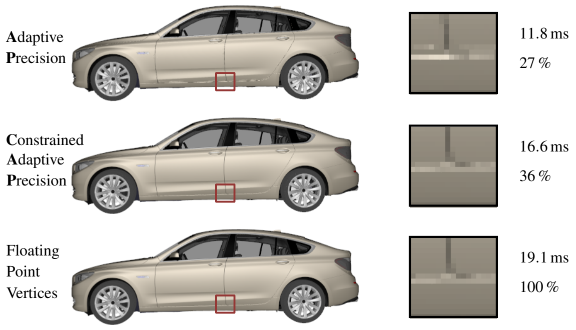


Figure 2: Comparing AP and CAP with floating-point vertex buffers. The timing values indicate the frame time. The percentage value represents the relative memory usage over floating-point vertex buffers. While AP causes rendering artifacts, CAP achieves quality indistinguishable from floating-point vertices, saves memory, and renders faster.

Adapting precision by simply cutting off trailing bits has the advantage that both compression and decompression are very efficient. Compression requires little preprocessing and can be performed on-the-fly, precision can be adapted efficiently, and decompression can be done in a vertex shader without negative impact on performance (in fact rendering from a compressed buffer is usually faster due to the reduced memory bandwidth). We will refer to this simple approach as *Adaptive Precision (AP)* in the following.

In our experience, AP often delivers surprisingly good results, but for highly detailed meshes shading artifacts occur when quantization becomes too coarse. We analyze this shading error due to vertex quantization, and show that we can constrain the error by restricting quantization for each vertex to a minimum bit number. As a result, vertices can have different quantization levels, which make decompression and compression a bit more complicated. However, it turns out, that also this *Constrained Adaptive Precision (CAP)* approach can be implemented efficiently, and very well handles shading, texture, and depth errors.

Figure 2 summarizes our results. AP allows for significant memory savings and high performance. Yet, artifacts are visible at large viewing distances (top row). With CAP, these artifacts disappear (middle row) while still keeping memory consumption and frame times low, and the rendering is indistinguishable from the original mesh (bottom row).

Both our approaches generate significant memory savings on their own, but we do not consider them as a real LOD-method (e.g., for coarse quantization, triangles degenerate to lines or points, and could be sorted out to reduce work load). Instead, we consider LOP to be best used in combination to *further compress* the vertex positions of a simplified mesh. LOP is extremely simple, very easy to integrate, and gives significant memory saving with only very little extra effort.

2. Previous Work

A widespread multi-resolution approach is the *progressive meshes (PMs)* technique [Hop96]. It is particularly suited for view-dependent refinement [Hop97]. In a preprocess Sander and Mitchel [SM05] create a discrete set of meshes with decreasing numbers of vertices and triangles using PMs. However, during runtime, vertex buffers of successive level-of-details need to reside on the GPU to allow for smooth LOD transitions. A first GPU implementation of continuous PMs consume 57 % more memory than an indexed face set [HSH09]. Derzapf et al.'s GPU implementations of PMs require 50 % the data of an indexed face set [DMG10b, DMG10a].

Vertex cluster techniques [RB93, Lin00] reduce the geometric complexity by partition the object's bounding geometry into uniform cells. All vertices of one cell are replaced by a single vertex representative. A GPU implementation for cluster creation exist [DT07] but they are not used to reduce the memory footprint during rendering.

Vertex quantization reduces complexity by discretizing the vertex coordinates to a grid that uniformly samples the mesh's bounding geometry. While early approaches determine the sample spaces, i.e., the number of bits, by empirical tests [Dee95], Chow [Cho97] proposes an iterative algorithm and assigns individual quantization levels for each vertex based on geometric criteria. However, shading error considerations are not examined. Computer graphics APIs like OpenGL 4.1 or Direct3D 11 support different quantization levels for vertices. However, the precision is fixed and cannot be altered dynamically. Purnomo et al. [PBCK05] quantize all vertex attributes to fit into a given bit-budget. The number of bits allocated by each attribute is optimized in a preprocess using a screen-space error metric. They decompress the attributes in a vertex shader however vertex precision may not be refined dynamically.

Vertex quantization has excessively been applied in the context of *geometry compression* [AG05]. They are often combined with variable bit-length codes. However, as the decompression algorithms are of sequential nature, they are difficult to implement on a parallel GPU architecture. GPU implementations exist only for special cases, such as terrain rendering [LC10] where geometry is represented by scalar valued height-maps rather than vector valued vertices.

Approaches to progressively transmit bits of vertex components [LK98] are designed for either on-disk storage or network transmissions and not for GPUs. Hao et al. [HV01] analyze the numerical error imposed by vertex transform operations and determine the accuracy in bits at which these operations have to be carried out, but do not leverage their result for reducing the memory footprint.

LOD-methods typically consider geometric error, assuming that a small geometric error also results in little rendering error. Appearance preserving LOD approaches optimize for the apparent rendering error [GH97, KSS98]. One key observation is that geometric error in highly curved surface regions translates to a large apparent error [Lin03, Cho97], so meshes should be preferably coarsened in flat regions.

3. Level-of-Precision

In order to adapt the precision level, we first quantize each coordinate using a 24-bit fix-point representation, which corresponds to unsigned floating-point precision [ILS05]. For each vertex only the b most significant bits of each component are tightly packed in video memory, where b defines the currently used precision. We refer to it as the current *bit level*. During rendering, the bits are *unpacked* for each vertex and converted back to floating-point. When more precision is required, additional bits are uploaded and merged with those already in video memory. Likewise, superfluous bits are removed if less precision is sufficient. For simplicity all vertex components, i.e., x, y, and z, use the same number of bits. Bits having the same bit-index i within the components of a vertex buffer are said to be on the *ith bit plane*.

Of course, a reduced LOP introduces apparent deviations in the rendered image. We look at a particular image sample, and identify three sources of error for its computed color:

- Coverage error: the coverage of the sample changes, i.e., the sample sees the wrong object. This error is most obvious along silhouettes.
- Attribute error: due to the modified vertex positions, wrong attribute values are used for shading. E.g., for highly varying shading normals, a highlight can wrongly appear or disappear at the sample. Similarly, a texture can be accessed at an offset position, yielding a wrong color.
- Depth error: the z order of nearby objects changes, so that a more distant object penetrates a closer one at the sample.

In Section 4 we introduce our *Adaptive Precision (AP)* algorithm: We select one common bit level for all vertices of a vertex buffer, such that the screen-space error is less than half a pixel. If the vertices of the buffer move, they are coarsened or refined by removing or adding bit planes. This allows balancing coverage error and memory savings efficiently.

However, particularly for highly-detailed models, attribute and depth errors occur once the bit level of the vertex buffer underflows a certain limit. As this generally only affects a small fraction of the vertices, we constrain those *critical* vertices, and select a minimum bit level for each of them. This minimum bit level may not be under-flown when precision is coarsened for the remaining vertices of the buffer. This approach – referred to as *Constrained Adaptive Precision (CAP)* – is discussed in Section 5.

4. Adaptive Precision

To save video memory, we choose the precision of the vertices, such that the coverage error is below a predefined threshold, e.g., half a pixel. We then store only the bits that are required for this precision in an *adaptive precision buffer*.

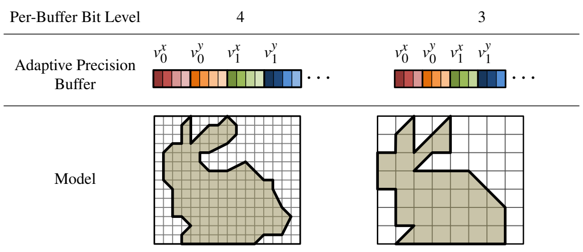

Per-Buffer Bit Level
4
3
Adaptive Precision Buffer
v_0^x v_0^y v_1^x v_1^y ...
v_0^x v_0^y v_1^x v_1^y ...
Model

Figure 3: 2D example of a vertex buffer stored in an adaptive precision buffer. The bits are represented by the colored bars. The 2D grid represents the possible quantization locations for the vertices. We dynamically adapt the number of bits of a vertex component by a screen-space criterion. To save memory, the bits are stored tightly without fragmentation.

The bits of the vertices are stored tightly packed, i.e., without internal fragmentation. All elements use the *same* number of bits, or the same *per-buffer bit level*. We determine it as the precision of the one bounding-box vertex closest to the camera. If the object moves, we adapt the precision by adding or removing entire bit planes to all the elements in the buffer: we temporarily *unpack* the vertex components, *add* or *remove* bits, and then *pack* them tightly again. Figure 3 illustrates the principle for an adaptive precision buffer with 3 and 4 bits per component. With this representation we obtain efficient random access, e.g., in a vertex shader (Section 6.2), during rendering.

Note that we have to provide video memory space to temporarily unpack the vertex buffer. If the model is too big it needs to be partition into smaller sub-models [Sha08]. When adapting the bit-level we process the vertex buffers of the sub-models sequentially.

If two adjacent vertex buffers use different bit levels, initially identical boundary vertices may be quantized to different positions, and unwanted cracks can occur (Figure 4 left). To avoid this, we compute a *per-vertex* bit level after vertex decoding, using the distance of the vertex to the camera. This per-vertex bit level is smaller or equal than the per-buffer bit level, and we zero out the superfluous least sig-

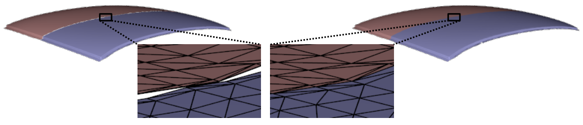


Figure 4: Removing cracks. When two vertex buffers possess different bit levels cracks may occur (left). We compute bit levels at vertex level and zero-out superfluous bits, and thus close the cracks (right).

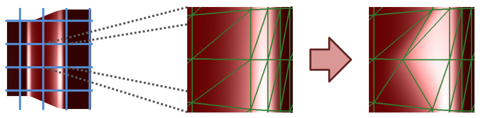


Figure 5: Shading error caused by a displaced vertex. The pixel raster grid is shown in blue.

nificant bits. This makes boundary vertices identical in both vertex buffers, and cracks are avoided (Figure 4 right).

5. Constrained Adaptive Precision

For highly detailed models at large viewing distances, attribute and depth errors become apparent when using AP. A solution would be to disallow the removal of bit planes below a certain bit level for *all* vertices. This, however, would dramatically limit the memory gains. Instead, we assign a minimum bit level that may not be under-flown *for every vertex individually*. During rendering, we generally use the per-buffer bit level, that avoids coverage errors, but if that is not precise enough to avoid attribute and depths errors for a particular vertex, we use its per-vertex minimum bit level. To this end we solve two problems: In a preprocess, we identify the minimum bit level for each vertex (Sections 5.1 and 5.2). At run-time, a fast and memory efficient data-structure (Section 5.3) has to account for the minimum per-vertex bit level of constrained adaptive precision data.

5.1. Attribute Error

An illustrative example for attribute error resulting from moving vertices due to reduced LOP is shown in Figure 5. In a region of high curvature a vertex is displaced to the left. As a result, the triangles on its right exhibiting a glossy highlight are significantly enlarged. Even if the displacement is far less than a pixel (pixel grid marked in blue), the average color of the pixel changes significantly. As a result the entire pixel color becomes much brighter, even with high-quality anti-aliasing. A similar effect is also visible in the upper right of Figure 2, where along the highly curved rims strong disturbing variations in brightness become visible.

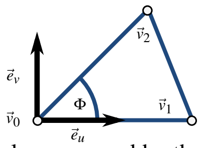
For the cause of this artifact, consider the two meshes projected onto a pixel grid in the top row of Figure 6. Due to quantization, the center vertex is moved to the right by less than a pixel (second row). We now look at the resulting shading error in image space, at the target position of the vertex (dashed line of last row). For the original mesh, the sample sees the interpolated normal (orange vector). Since the normal attribute remains unchanged when the vertex is moved, the sample sees the original vertex normal (red vector) when rendering the quantized mesh. This difference between the two normals is low for the low curvature mesh on the left, and high for the high curvature mesh on the right.

Note that the vertex normal is an attribute, which is computed for the original mesh and then remains unchanged after quantizing vertices. This means that if due to quantization T-vertices or even fold-overs appear, the normals used for lighting is neither undefined nor flips. Thus the only source for rendering errors is that wrong arguments are used for the shader call as described above. Fold-overs are generally no problem in our scenario. Noteworthy exceptions are transparent surfaces rendered with alpha-blending, because here the overlapping triangles result in increased opacity.

Next, we quantify the error shown in Figure 6. We consider a triangle with vertices $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$. With each vertex an attribute vector $\vec{a}_i$ is stored (shading normal, texture coordinate, etc.). During rasterization, barycentric interpolation is used for determine the attribute of a raster sample

$$\vec{a}_{\text{bary}}(\beta,\gamma) = (1-\beta-\gamma)\vec{a}_0 + \beta\vec{a}_1 + \gamma\vec{a}_2 = \vec{a}_0 + \mathbf{A}\begin{bmatrix}\beta\\ \gamma\end{bmatrix},$$

where $(1-\beta-\gamma,\beta,\gamma)$ are the barycentric coordinates at the raster sample and $\mathbf{A} = \begin{bmatrix}\vec{a}_1-\vec{a}_0 & \vec{a}_2-\vec{a}_0\end{bmatrix}$. The pixel shader f uses this interpolated attribute to compute the sample's color $f(\vec{a}_{\text{bary}}(\beta,\gamma))$.



Next, we bound the difference in the sample's color due to the quantization movement of one vertex. Without loss of generality, we consider $\vec{v}_0$. We assume that $\vec{v}_0$ moves along the tangent plane spanned by the triangle. In order to correctly measure the movement of $\vec{v}_0$, we have to express it in coordinates (u,v) that are defined in a basis $\{\vec{e}_u,\vec{e}_v\}$ that is orthonormal in world space (see inset):

$$\begin{bmatrix}u\\ v\end{bmatrix} = \begin{bmatrix}\|\vec{v}_1-\vec{v}_0\| & \|\vec{v}_2-\vec{v}_0\|\cos\phi\\ 0 & \|\vec{v}_2-\vec{v}_0\|\sin\phi\end{bmatrix}\begin{bmatrix}\beta\\ \gamma\end{bmatrix} = \mathbf{M}\begin{bmatrix}\beta\\ \gamma\end{bmatrix},$$

where ϕ is the triangle's angle at $\vec{v}_0$. Using $\mathbf{M}$, we can compute the attribute value at (u,v) as

$$\vec{a}_{\text{orth}}(u,v) = \vec{a}_{\text{bary}}\left(\mathbf{M}^{-1}\begin{bmatrix}u\\ v\end{bmatrix}\right) = \vec{a}_0 + \mathbf{A}\mathbf{M}^{-1}\begin{bmatrix}u\\ v\end{bmatrix}.$$

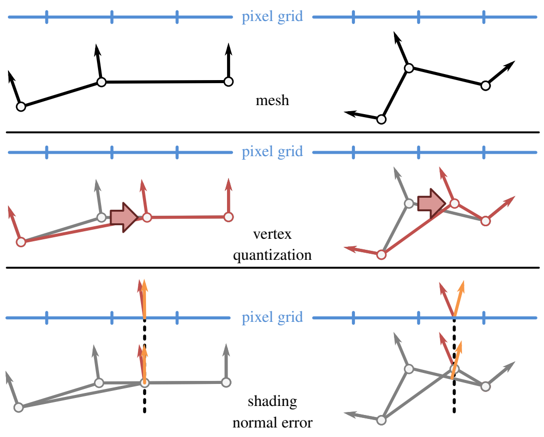


Figure 6: Moving a vertex of a flat (left) and curved mesh (right) results either in weak (left) or strong (right) variation of the shading normal.

When moving $\vec{v}_0$ by some offset (u,v), the attribute $\vec{a}_0 = \vec{a}_{\text{orth}}(0,0)$ is used at the target position, instead of the correctly interpolated value $\vec{a}_{\text{orth}}(u,v)$, which causes the error

$$E = \|f(\vec{a}_{\text{orth}}(u,v)) - f(\vec{a}_0)\|.$$

Linearly approximating $f(\vec{a}_{\text{orth}}(u,v))$ by a Taylor expansion using the Jacobian $\mathbf{J}_f$ of f

$$f(\vec{a}_{\text{orth}}(u,v)) \approx f(\vec{a}_0) + \mathbf{J}_f \mathbf{A}\mathbf{M}^{-1} \begin{bmatrix} u \\ v \end{bmatrix},$$

gives us an error estimate

$$E \approx \left\| \mathbf{J}_f \mathbf{A}\mathbf{M}^{-1} \begin{bmatrix} u \\ v \end{bmatrix} \right\| \leq \underbrace{\|\mathbf{J}_f\|}_{k_f} \underbrace{\left\|\mathbf{A}\mathbf{M}^{-1}\right\|}_{E_t} \underbrace{\left\| \begin{bmatrix} u \\ v \end{bmatrix} \right\|}_{\Delta x}.$$

As vector and matrix norms, we use the Euclidean and spectral norm, respectively. Δx is the magnitude by which a vertex moves due to quantization. E_t measures, how fast the attribute varies over the triangle, so it only depends on the triangle and its attributes. k_f depends on the shader, and defines the maximum slope of the shader result with respect to the attribute $\vec{a}$. If $\vec{a}_i$ are normals used as input to a Phong shader, k_f equals the specular coefficient times the specular exponent. If $\vec{a}_i$ are texture coordinates, k_f is the maximum slope of the texture signal.

Up to now, we only considered a single triangle, but we want error bounds per vertex v. We thus compute the maximum value for all surrounding triangles of a vertex v, and call this value E_v. Putting everything together, we bound the shading error when moving vertex v by Δx by

$$k_f\, E_v\, \Delta x \leq \varepsilon,$$

where ε is the maximum color intensity error. To be below that error, we limit the magnitude Δx by which the vertex moves due to quantization. This immediately leads to a *per-vertex minimum bit level* b, by setting $\Delta x = 2^{-b}$ and solving for b. A visualization of our error analysis considering the normal attribute is shown in Figure 7. For another result of the constrained quantization refer to Figure 2.

5.2. Depth Error

A reduced LOP can also result in depth errors as illustrated in Figure 8. The blue and the red curve in (a) and (b) depict

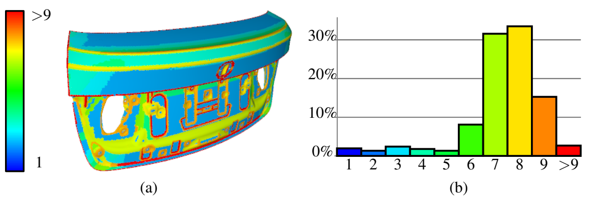


Figure 7: Per-vertex minimum number of bits determined by our shading error analysis. (a) The vertices are color-coded by their minimum bit level. (b) Histogram of the relative amount of vertices for each minimum bit level.

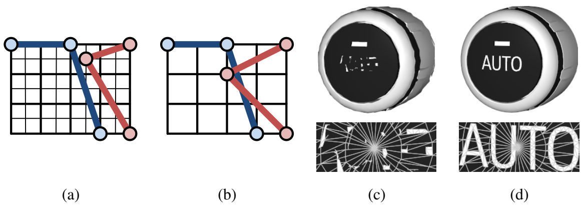


Figure 8: Depth artifacts. (a, b) The blue and the red curve depict two objects that change their order once their bit level decreases. (c, d): The button and its labeling are two different objects. While AP cannot handle correct ordering of the objects (c), constraining the bit level of depth-critical vertices using CAP removes depth artifacts (d).

two different objects. At the higher bit level, the blue object is entirely in front of the red one (a). When the bit level decreases, the objects intersect (b), resulting in depth artifacts (c). By constraining the minimum bit levels of those depth-critical vertices, we can guarantee a correct order of the objects and avoid rendering artifacts (d). For now we use a simple $\mathcal{O}(n^2)$ algorithm to determine depth-critical vertices and fix their minimum bit level to a user defined value.

5.3. Binned Adaptive Precision Buffers

With CAP we essentially need to store an individual number of bits per vertex. However, explicitly storing a bit level with every vertex would result in additional memory overhead. Instead, we use a *binned adaptive vertex buffer*: we sort the vertices into *bins* according to their per-vertex minimum bit levels. During runtime we add necessary bit planes to bins with a bit level smaller than the view-dependent per-buffer bit level, and tightly pack the bits, as illustrated in Figure 9.

Our data structure allows easy random access: To *unpack* the jth vertex we determine the bin $b-1$ it resides in, using a small look-up table containing the first vertex index of each bin, `VertexId`, and its corresponding bit index, `BitId`. We then read $3 \cdot b$ bits from the bit index $3 \cdot b \cdot (j - \texttt{VertexId}[b-1]) + \texttt{BitId}[b-1]$. The look-up table has a small and constant size: the maximum number of bits per component,

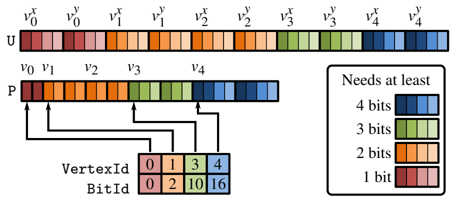

v_0^x v_0^y v_1^x v_1^y v_2^x v_2^y v_3^x v_3^y v_4^x v_4^y
U
v_0 v_1 v_2 v_3 v_4
P
VertexId 0 1 3 4
BitId 0 2 10 16
Needs at least
4 bits
3 bits
2 bits
1 bit

Figure 9: 2D example of a binned adaptive precision buffer. The vertices of the original vertex buffer `U` are sorted by their bit level in ascending order. Only bits above the minimum bit level are initially packed into the binned adaptive precision buffer `P`. The table `VertexId` marks the bit level boundaries and the entries of `BitId` hold their bit indices.

e.g., 24 in our case. Therefore, we can hard-code a binary search in a few instructions without branches. Note that bin b goes from `VertexId`$[b]$ to `VertexId`$[b+1]-1$, and if `VertexId`$[b]$ = `VertexId`$[b+1]$, the bin is empty.

Similar to adaptive precision buffers, we are able to adjust the per-buffer bit level: We temporarily unpack all the vertices and *add* or *remove* entire bit planes. However, we need to take care that we keep the minimum bit level for each bin. Finally, we adjust `VertexId` and `BitId` and tightly *pack* the vertices again, as shown in Figure 10.

6. Data-Parallel Implementation

GPU vertex buffers stored in (binned) adaptive precision buffers have several advantages: They contain only the precision that is currently needed, saving precious GPU memory. When a bit is added, only the associated bit plane has to be supplied, which reduces the amount of data transferred to the GPU. Further, removing a bit does not even require any CPU-GPU data transfers at all. First, we explain the main operations for (binned) adaptive precision buffers. Then we describe how easily they are integrated into shader programs.

6.1. Data-Parallel Precision Adaption

To adapt the precision, we *unpack* the *adaptive precision buffer* into a temporary buffer. For each element we use one thread, gather multiple system words from the adaptive precision buffer, and combine them to one unpacked element. Bits not specified by the current bit level are set to zero (Figure 11a). Similarly, we use one thread per system word to tightly *pack* the elements of the temporary buffer again (Figure 11b). If required, the *remove-bit* operation zeros out ad-

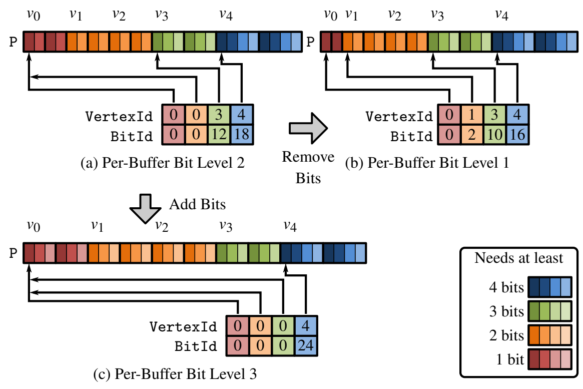


Figure 10: 2D example of precision adaption for a binned adaptive precision buffer. (a) Every vertex of the binned adaptive precision buffer `P` has at least 2 bits. (b) When lowering the per-buffer bit level only red bits may be removed. (c) When increasing the per-buffer bit level from 2 to 3 bits red and orange vertices receive new bits. Note how `VertexId` and `BitId` adapt with changing bit levels.

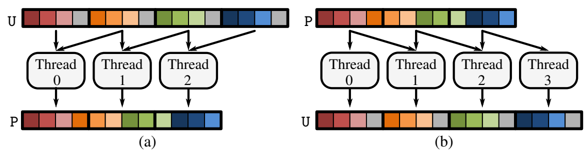


Figure 11: Unpack and unpack operation. The adaptive precision buffer `P` has bit level 3; the system word size is 4. (a) By gathering from `P`, the ith thread unpacks one element to `U[i]`. (b) Elements from the unpacked buffer `U` are gathered by the ith thread and written to `P[i]`.

ditional bits before packing. If we wish to *add* a bit, we upload the new bit plane into a separate GPU buffer, and each thread merges its new bit with its unpacked element.

The operations for *binned adaptive precision buffers* operate in the same way. The most noteworthy differences are that we need to provide the look-up tables `VertexId` and `BitId` for packing and unpacking, and take care of the per-vertex minimum bit level in each bin.

6.2. Graphics Pipeline Integration

Adaptive precision buffers containing vertex positions can be integrated into the programmable stages of the GPU pipeline efficiently, e.g., in the vertex shader: Attribute and index buffers are bound as usual. The adaptive precision buffer with vertex positions is bound as texture. In each vertex shader thread, we unpack the x, y, and z components from the adaptive precision buffer and convert them to floating-point numbers. Afterwards, the vertex shader proceeds as usual. For binned adaptive precision buffers we provide `VertexId` and `BitId` as uniform variables.

7. Results and Discussion

To evaluate our AP (Section 4) and CAP (Section 5) algorithms, we use two indexed triangle sets: an industry model of a car (compare Figure 2) and the triangulated model of Michelangelo's David from the Digital Michelangelo Project [LPC*00]. Each vertex comes with one additional shading normal. The car consists of 7.3 M vertices scattered across 1981 sub-models, whose sizes range from 4 to 330 k vertices. The original David data set is assembled from 150 sub-models and consists of nearly 1 billion polygons. We decimate them using vertex quantization [GH97] to fit into one GiB including per-vertex normals. After decimation, the model has 20 M vertices. All experiments are carried out on an Nvidia GeForce GTX 480 with 1.5 GiB video memory at a resolution of 1280×720 with $16\times$ MSAA using OpenGL 4.1. We reserve temporary memory required for bit level adaption to fit the largest sub-model. Bit level adaption is implemented using vertex shaders and transform feedback. We set the error-tolerance to half a pixel. For CAP we em-

Table 1: Frame times in milliseconds and relative memory consumption for our test scenes at different average bit levels. The scenes are position relative to the camera such that bit levels demanded by our screen-space criterion go from 6 to 11 bits (column **BL**). Columns **AP** and **CAP** refer to the algorithms of Section 4 and 5, respectively. Percentage values in parenthesis denote the memory usage relative to standard vertex buffers containing three floating-point values per vertex. For comparison, timings in columns **UC** are measured with uncompressed floating-point vertex buffers.

	Car			**David**		
BL	**AP**	**CAP**	**UC**	**AP**	**CAP**	**UC**
6	10.8 (19%)	14.9 (29%)	18.5	24.2 (19%)	28.8 (24%)	75.9
7	11.0 (22%)	15.6 (30%)	18.9	24.2 (22%)	28.4 (26%)	75.3
8	11.7 (25%)	16.7 (32%)	19.1	24.2 (25%)	29.8 (28%)	70.1
9	13.9 (28%)	19.3 (34%)	20.9	24.7 (28%)	41.2 (32%)	59.2
10	17.2 (31%)	21.5 (36%)	22.0	32.0 (31%)	40.8 (34%)	40.3
11	18.0 (34%)	20.7 (38%)	19.5	31.6 (34%)	32.7 (38%)	31.7

pirically choose k_f to avoid shading artifacts and turn off our depth-error constraint.

7.1. Rendering Performance and Memory Usage

We list rendering performance and memory usage at various bit levels in Table 1. Note that timings for bit-level changes are discussed separately in the next section. As the bit level depends on the distance of the model to the camera, we vary it such that the average bit level of all sub-models ranges between 6 and 11 bits. We compare the time per frame between AP, CAP, and uncompressed vertex buffers with single precision floating-point elements (columns UC). Note that the frame-times in columns UC of the David model are higher the further the object is away from the camera. This is because more and more triangles compete for the same pixels, which stresses the GPU's z-buffer. For models filling the entire screen, an average of 10 bits in precision is required. This reduces memory consumption for vertices to *one third*, while still maintaining high rendering performance. Note that due to the reduced bandwidth requirements AP is always faster than regular floating-point vertex buffers. As depicted in Figure 2, shading artifacts are visible for AP, but can be avoided using CAP. Even though memory access is more complex for CAP frame times are generally faster than standard vertex buffers, while still saving more than 60 % memory.

7.2. Changing Bit Levels

In Figure 12, we separate the timings for the car model into the time spent for rendering (blue curves) and for adapting bit levels (red curves). The latter include the time for packing, unpacking, resizing buffer objects, and uploading additional bits. We test both AP (left column) and CAP (right column) using two typical camera motions for object visualization: For a *rotation* of the car around its vertical axis (top row) we observe an average per-buffer bit level of 9.4 to 9.8 per frame. Further, in a *dolly motion*, we move the car towards the camera and then back again (bottom row). Here, the average per buffer bit level varies from 6.0 to 10.9. Generally, AP exhibits higher performance than CAP. This is due to the fact that more data is transferred, and both pack- and unpack operations are more sophisticated.

During the rotational motion only a few vertex buffers require bit level adaption, making the impact of bit level changes negligible. The dolly motion also reveals that changing bit levels is a rare event, and it does not affect the overall performance much. However, peaks appear, when many small buffers change their levels during one frame, causing substantial setup overhead. For the car model these rare peaks make up about 50 % of the total frame time. When applying the same dolly motion to the David model, which possesses fewer vertex buffers, bit level adaption consumed at most 32 % of the frame time. Whenever bits are added, the time spent for uploading them only accounts for about one tenth of the entire add-bit operation.

7.3. Quality

To evaluate image quality, we render the back lid of the car. It has strong creases which are particularly sensitive to shading errors. We subtract the images generated with AP (a) and CAP (b) from the reference image generated with floating-point vertex buffers, and display the intensity differences as heat map in Figure 13. For AP we observe intensity errors along creases. By constraining the bit level using the preprocess of Section 5.1 and CAP, we are able to reduce those significantly, while still keeping memory requirements low.

The car model exhibited perceivable popping artifacts at bit level changes when using AP. These artifacts disappeared when using CAP and our attribute error analysis from Section 5.1. In contrast to the car, David was less sensitive to popping artifacts when using AP only, and no shading problems were encountered, either. Given that AP requires less memory and renders faster than CAP, it is preferable for

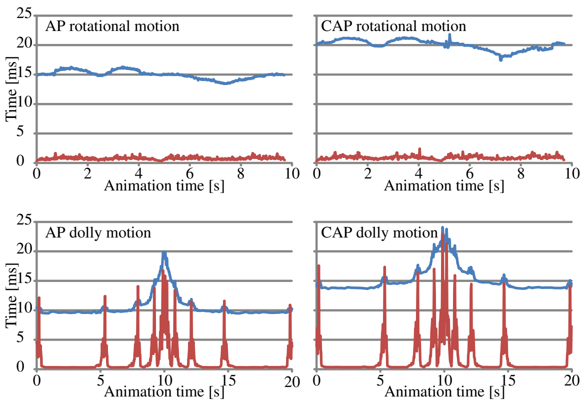


Figure 12: Time for changing bit levels (red curve) and rendering (blue curve). The top row shows the result for a rotational motion, the bottom row for a dolly motion.

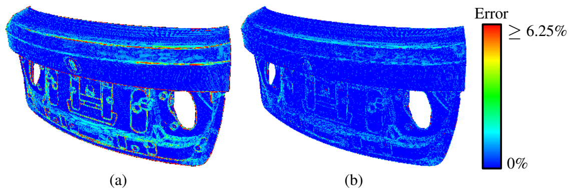


Figure 13: Heat-map coded difference images of renderings generated with our algorithms and the ground truth using floating-point vertices. The error is measured with respect to the maximum intensity value of each pixel. (a) AP: PSNR 29 dB, Mem. 28 % (b) CAP: PSNR 39 dB, Mem. 38 %.

similar types of models, or when shading, texture, or depth errors are tolerable.

8. Conclusion and Future Work

We presented a level-of-precision approach for the compression of vertex position data in GPU memory. We introduced *Adaptive Precision (AP)*, handling coverage error, and *Constrained Adaptive Precision (CAP)*, reducing shading, texture, and depth error. Both representations generally render faster than standard floating-point vertex buffers. Our methods allow for refining and coarsening the level of precision by adding or removing bit planes interactively. For our test scenes, we typically save between 62 and 81 % of the vertex data at little or no loss in rendering fidelity.

We have tested our level-of-precision approach on vertex *positions* only. Vertex attributes must be considered separately: For instance, normal vectors should not be stored with varying precision, because this would result in changing color values. Static compression methods are preferable [MSS*10]. For texture coordinates a compression scheme similar to the one for vertices might be reasonable, but we have not examined this yet.

Our approach can be applied on top of classical LOD-approaches. It works directly for discrete LOD, but a continuous adaption of the precision level only makes sense if few discrete LOD are used. Our method can also be applied in combination with PMs. Using half-edge collapses every vertex split only requires uploading a single new vertex, which makes a simple integration of LOP possible. CAP requires that newly integrated vertices are inserted at the correct positions, which means to rearrange the adaptive precision buffers. Moreover, adaptive LOP is not limited to polygonal meshes: terrain, point-based, and parametric surfaces rendering are applications which we want to investigate.

References

[AG05] Alliez P., Gotsman C.: Recent advances in compression of 3D meshes. In *Advances in Multiresolution for Geometric Modelling*. Springer, 2005, pp. 3–26.

[Cho97] Chow M. M.: Optimized geometry compression for real-time rendering. In *Proc. of VIS'97* (1997), pp. 346–354.

[Dee95] Deering M. F.: Geometry compression. In *Proc. of SIGGRAPH'95* (Aug. 1995), pp. 13–20.

[DMG10a] Derzapf E., Menzel N., Guthe M.: Parallel view-dependent out-of-core progressive meshes. In *Proc. of VMV'10* (2010), pp. 25–32.

[DMG10b] Derzapf E., Menzel N., Guthe M.: Parallel view-dependent refinement of compact progressive meshes. In *Proc. of EGPGV'10* (2010), pp. 53–62.

[DT07] DeCoro C., Tatarchuk N.: Real-time mesh simplification using the GPU. In *Proc. of I3D'07* (2007), pp. 161–166.

[GH97] Garland M., Heckbert P. S.: Surface simplification using quadric error metrics. In *Proc. of SIGGRAPH'97* (1997), pp. 209–216.

[Hop96] Hoppe H.: Progressive meshes. In *Proc. of SIGGRAPH'96* (1996), pp. 99–108.

[Hop97] Hoppe H.: View-dependent refinement of progressive meshes. In *Proc. of SIGGRAPH'97* (1997), pp. 189–198.

[HSH09] Hu L., Sander P. V., Hoppe H.: Parallel view-dependent refinement of progressive meshes. In *Proc. of I3D'09* (2009), pp. 169–176.

[HV01] Hao X., Varshney A.: Variable-precision rendering. In *Proc. of I3D'01* (2001), pp. 149–158.

[ILS05] Isenburg M., Lindstrom P., Snoeyink J.: Lossless compression of predicted floating-point geometry. *Computer-Aided Design 37*, 8 (2005), 869–877.

[KSS98] Klein R., Schilling A., Strasser W.: Illumination dependent refinement of multiresolution meshes. In *Proc. of CGI'98* (1998), pp. 680–687.

[LC10] Lindstrom P., Cohen J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *Proc. of I3D'10* (2010), pp. 65–73.

[Lin00] Lindstrom P.: Out-of-core simplification of large polygonal models. In *Proc. of SIGGRAPH'00* (2000), pp. 259–262.

[Lin03] Lindstrom P.: Out-of-core construction and visualization of multiresolution surfaces. In *Proc. of I3D'03* (2003), pp. 93–102.

[LK98] Li J., Kuo C.-C.: Progressive coding of 3D graphic models. *Proc. of the IEEE 86*, 6 (1998), 1052 –1063.

[LPC*00] Levoy M., Pulli K., Curless B., Rusinkiewicz S., Koller D., Pereira L., Ginzton M., Anderson S., Davis J., Ginsberg J., Shade J., Fulk D.: The Digital Michelangelo Project: 3D scanning of large statues. In *Proc. of SIGGRAPH '00* (2000), pp. 131–144.

[MSS*10] Meyer Q., Süssmuth J., Sussner G., Stamminger M., Greiner G.: On floating-point normal vectors. *Computer Graphics Forum 29*, 4 (2010), 1405–1409.

[PBCK05] Purnomo B., Bilodeau J., Cohen J. D., Kumar S.: Hardware-compatible vertex compression using quantization and simplification. In *Proc. of Graphics Hardware'05* (July 2005), pp. 53–62.

[RB93] Rossignac J., Borrel P.: Multi-resolution 3D approximations for rendering complex scenes. In *Geometric Modeling in Computer Graphics*. Spinger, 1993, pp. 455–465.

[Sha08] Shamir A.: A survey on mesh segmentation techniques. *Computer Graphics Forum 27*, 6 (2008), 1539–1556.

[SM05] Sander P. V., Mitchell J. L.: Progressive buffers: View-dependent geometry and texture for LOD rendering. In *Proc. of SGP'05* (July 2005), pp. 129–138.