2

Real-Time Deformable Terrain Rendering with DirectX 11

Egor Yusov

2.1 Introduction

High geometric fidelity real-time terrain visualization is a critical part of many applications such as flight simulators and computer games. For a large data set, a brute-force approach cannot be exploited even on the highest-end graphics platforms due to memory and computational-power limitations. Thus, it is essential to dynamically control the triangulation complexity and reduce the height-map size to fit the hardware limitations and meet real-time constraints.

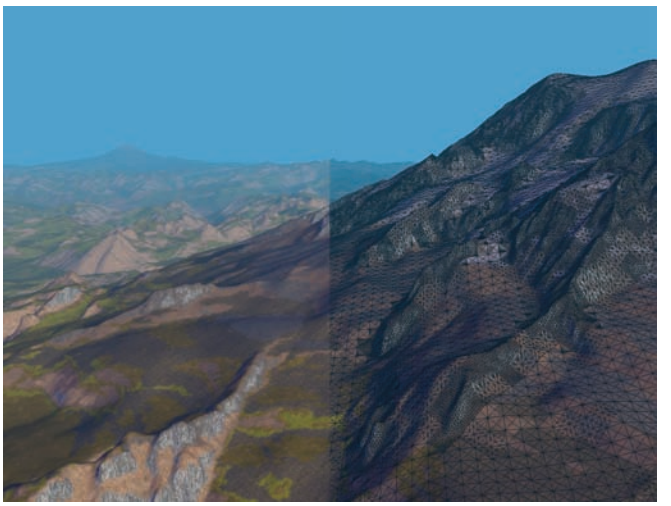


Figure 2.1. Puget Sound elevation data set rendered with the proposed method.

To effectively render large terrains, many dynamic multiresolution approaches have been developed in the past years. These algorithms typically adapt the terrain tessellation using local surface-roughness criteria together with the view parameters. This allows dramatic reduction of the model complexity without significant loss of visual accuracy. New capabilities of DX11-class graphics hardware enable a new approach, when adaptive terrain tessellation is built entirely on the GPU using the dedicated hardware unit. This increases the triangle throughput and reduces the memory-storage requirements together with the CPU load. It also reduces the amount of data to be transferred from the main memory to the GPU that improves rendering performance as well.

To reduce the storage requirements, several recent approaches exploit compression schemes. Examples of such methods are geometry clipmaps [Losasso and Hoppe 04] and C-BDAM [Gobbetti et al. 06]. Though these methods are optimized for maximum GPU efficiency, they completely ignore local terrain surface features, significantly increasing the GPU load, and do not support real-time deformations.

Dynamic terrain modifications are an important feature for a number of applications such as games, where the action changes the terrain, landscape editors, and other elements. It poses a number of problems such as the need to construct new triangulation and update internal data structures, which are especially complex when compressed representation is exploited. As a result, real-time terrain deformations are usually not considered.

The new terrain-rendering technique presented in this chapter combines an efficient compression scheme with the GPU-accelerated triangulation construction method and, at the same time, supports dynamic modifications. The technique has the following advantages:

- The proposed multiresolution compressed height-map representation substantially reduces the storage requirements and enables direct control of a reconstruction precision.
- Height-map decompression is accelerated by the GPU, which reduces expensive CPU-GPU data-transfer overhead and eliminates data duplication in main and GPU memory.
- Triangulation construction is completely done by the tessellation unit of recent graphics hardware:
  - Topology is not encoded and completely generated by the GPU for each camera position.
  - Strict screen space error bound of the rendered surface is provided.
  - The topology is updated automatically as the terrain is modified.
- The technique is efficiently implemented using hardware-supported features such as texture arrays and instancing.

- The terrain is fully dynamic; all modifications are permanent and all affected regions are updated in the compressed representation.

2.2 Algorithm Overview

At the preprocess step, the initial height map is converted to a quadtree data structure. To reduce the storage requirements, the hierarchy is encoded using a simple and efficient compression algorithm described in Section 2.3.

At run time, an unbalanced patch quadtree is maintained in decompressed form. It defines the coarse block-based terrain approximation. Each patch in this approximation is precisely triangulated using the hardware, such that the simplified triangulation tolerates the given screen space error bound. For this purpose, each patch is subdivided into equal-sized small blocks that are independently triangulated by the GPU-supported tessellation unit, as described in Section 2.4.

Procedural terrain texturing presented in Section 2.5 provides high-quality terrain surface texture with minimal memory requirements. Dynamic terrain modifications described in Section 2.6 consists of two parts. The first part is instant and is done on the GPU. The second part is asynchronous and is performed by the CPU: each modified terrain region is read back to the main memory and re-encoded. To achieve high rendering performance, a number of technical tricks described in Section 2.7 are exploited. Performance results are given in Section 2.8 and Section 2.9 concludes the paper.

2.3 Compressed Multiresolution Terrain Representation

2.3.1 The Patch Quadtree

Our multiresolution terrain model is based on a quadtree of equal-sized square blocks (hereinafter referred to as *patches*), a widely used structure in real-time terrain rendering (Figure 2.2). Each patch is $(2^n+1)\times(2^n+1)$ in size (65×65, 129×129, 257×257 etc.) and shares a one-sample boundary with its neighbors to eliminate cracks. We denote the patch located in level l $(0\le l\le l_0)$ at node (i, j) $(0\le i,j<2^l)$ by $H_{i,j}^{(l)}$. The sample located at column c and row r of the patch $H_{i,j}^{(l)}$ is denoted by $h_{c,r}^{(l)}$:

$$H_{i,j}^{(l)}=(h_{c,r}^{(l)}), 0\le c,r\le 2^n.$$

Note that $h_{c,r}^{(l)}$ notation will always refer to the sample of the patch $H_{i,j}^{(l)}$ so indices i, j are implied, but not included in the notation to simplify it. Quadtree construction is performed in a bottom-up order starting from level $l=l_0$ as follows:

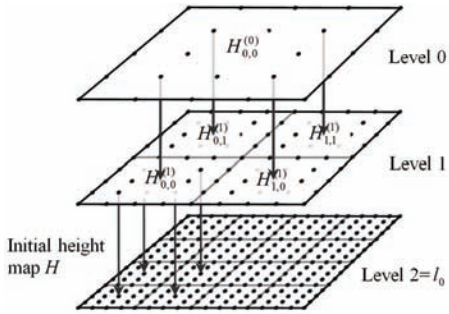


Figure 2.2. A three-level patch quadtree over the 17 × 17 input height map. The patch size is 5 ×5 ($n = 2$). (Courtesy of WSCG.)

1. Initial height map H is subdivided into $(2^n + 1) \times (2^n + 1)$ overlapping square blocks, which constitute patches $H_{i,j}^{(l_0)}$ at the finest resolution level $l = l_0$: $h_{c,r}^{(l_0)} = h_{2^n i+c,2^n j+r}$, $0 \le c, r \le 2^n$, where $h_{c,r}$ denotes the input height-map sample located at column c and row r.

2. Patches $H_{i,j}^{(l)}$ at coarse levels $l < l_0$ are derived from their children with the following steps:

   (a) Patch offspring are combined into a single $(2 \cdot 2^n + 1) \times (2 \cdot 2^n + 1)$ matrix $O_{i,j}^{(l)} = (o_{c,r}^{(l)})$, $0 \le c, r \le 2 \times 2^n$:

   $$O_{i,j}^{(l)} = H_{2i,2j}^{(l+1)} \cup H_{2i+1,2j}^{(l+1)} \cup H_{2i,2j+1}^{(l+1)} \cup H_{2i+1,2j+1}^{(l+1)};$$

   (b) The matrix $O_{i,j}^{(l)}$ is decreased by rejecting every odd column and row as shown in Figure 2.3: $h_{c,r}^{(l)} = o_{2c,2r}^{(l)}$, $0 \le c, r \le 2^n$. We denote this operation by $\downarrow_{2,2}$ (): $H_{i,j}^{(l)} = \downarrow_{2,2} (O_{i,j}^{(l)})$.

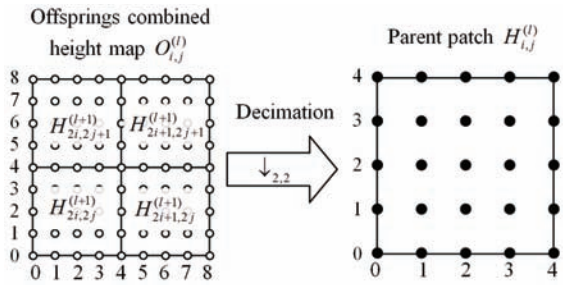


Figure 2.3. Deriving the parent patch height map from its offspring.

Quadtree construction stops at the coarsest resolution level (l=0) represented by the single patch $H_{0,0}^{(0)}$ covering the entire terrain.

2.3.2 The Compression Algorithm

To reduce memory overhead, the multiresolution hierarchy is maintained in a compressed form. The compression is done directly during the quadtree construction process. We exploit a simple yet quite efficient compression scheme that is derived from our previous work [Yusov and Shevtsov 11]. The proposed technique has the following key properties:

1. Reconstruction error is precisely bounded by the specified error tolerance.
2. Decompression is accelerated by the GPU.
3. A single pass bottom-up compression order enables dynamic height-map deformations. After the patch at the finest resolution level is modified, all updates can be propagated to coarser levels.
4. Patches from different terrain regions are processed independently and can be compressed/decompressed in parallel.

Quantization. During the compression, we input 16-bit height-map samples (which constitute patches $H_{i,j}^{(l_0)}$ at the finest resolution level $l = l_0$) to be quantized. Given a user-defined maximum reconstruction-error threshold $\delta \geq 0$, the following uniform quantizer with a dead zone is applied to each sample:

$$\tilde{h}_{c,r}^{(l_0)} = \left\lfloor (h_{c,r}^{(l_0)} + \delta)/(2\delta + 1) \right\rfloor \cdot (2\delta + 1), \quad 0 \leq i, j < 2^{l_0}, 0 \leq c, r \leq 2^n,$$

where $\lfloor x \rfloor$ rounds to the largest integer that is less than or equal to x; $h_{c,r}^{(l_0)}$ and $\tilde{h}_{c,r}^{(l_0)}$ are exact and quantized samples, respectively.

Quantized patches will be further denoted by $\tilde{H}_{i,j}^{(l)}$. Each quantized height $\tilde{h}$ is identified by an integer value $q = Q(h)$ where $Q(h) = \lfloor (h + \delta)/(2\delta + 1) \rfloor$. Knowing q, the decoder reconstructs $\tilde{h}$ as follows:

$$\tilde{h} = (2\delta + 1) \cdot q.$$

This quantization rule assures that the maximum reconstruction error is bounded by δ:

$$\max_{0 \leq c,r \leq 2^n} |\tilde{h}_{c,r}^{(l_0)} - h_{c,r}^{(l_0)}| \leq \delta, \quad 0 \leq i, j < 2^{l_0}.$$

Quantized patches $\tilde{H}_{i,j}^{(l)}$ at coarse levels $l < l_0$ are derived from their offspring in the same way as described in Section 2.3.1.

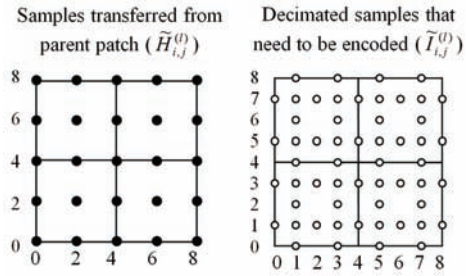


Figure 2.4. (a) Parent patch $\tilde{H}^{(l)}_{i,j}$ samples that are transferred to $\tilde{O}^{(l)}_{i,j}$. (b) Samples missing in $\tilde{O}^{(l)}_{i,j}$ that need to be encoded.

Encoding. Let us consider a patch's quantized height map $\tilde{H}^{(l)}_{i,j}$ and its offsprings' quantized combined height map $\tilde{O}^{(l)}_{i,j}$. Since decompression is performed in a top-down order, we can assume that $\tilde{H}^{(l)}_{i,j}$ is already decoded. Thus, samples located at even columns and rows in $\tilde{O}^{(l)}_{i,j}$ (Figure 2.4(a)) can be simply copied from $\tilde{H}^{(l)}_{i,j}$. The remaining samples (Figure 2.4(b)) were removed during the decimation and constitute refinement data that is required to completely reconstruct $\tilde{O}^{(l)}_{i,j}$. We denote these samples by $\tilde{I}^{(l)}_{i,j}$: $\tilde{I}^{(l)}_{i,j} = \tilde{O}^{(l)}_{i,j} / \tilde{H}^{(l)}_{i,j}$.

Encoding samples from $\tilde{I}^{(l)}_{i,j}$ consists of the following steps:

1. The sample value is predicted from the neighboring parent patch samples;
2. The predicted value is quantized;
3. The prediction error is calculated as the difference between exact quantized and predicted quantized values;
4. The error is encoded using arithmetic coding.

The prediction error is calculated for each $\tilde{h}^{(l)}_{c,r} \in \tilde{I}^{(l)}_{i,j}$ according to the following expression:

$$d^{(l)}_{c,r} = Q[\, P(\tilde{h}^{(l)}_{c,r})\,] - q^{(l)}_{c,r},$$

where $d^{(l)}_{c,r}$ is the prediction error for the sample $\tilde{h}^{(l)}_{c,r} \in \tilde{I}^{(l)}_{i,j}$, $P(\tilde{h}^{(l)}_{c,r})$ is the prediction operator for the sample, and $q^{(l)}_{c,r} = Q(\tilde{h}^{(l)}_{c,r})$ is the exact quantized value. We denote the prediction-error matrix for patch $\tilde{H}^{(l)}_{i,j}$ by $D^{(l)}_{i,j}$.

For the sake of GPU acceleration, we exploit the bilinear predictor $P_{BiL}(\tilde{h}^{(l)}_{c,r})$ that calculates a predicted value of $\tilde{h}^{(l)}_{c,r} \in \tilde{I}^{(l)}_{i,j}$ as a weighted sum of four nearest samples from $\tilde{H}^{(l)}_{i,j}$. A 1D illustration for this predictor is given in Figure 2.5.

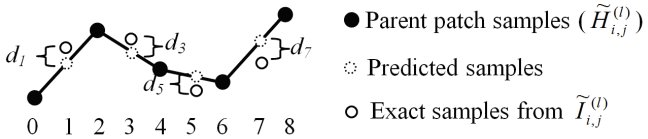


Figure 2.5. Linear prediction of missing samples and prediction errors (quantization of predicted values is not shown).

The magnitudes $|d_{c,r}^{(l)}|$ and signs $\text{sign}(d_{c,r}^{(l)})$ of the resulting prediction errors $d_{c,r}^{(l)}$ are encoded separately using adaptive arithmetic coding [Witten et al. 87] and output to the final encoded bit stream. We use an adaptive approach that learns the statistical properties of the input symbol stream on the fly. This is implemented as a histogram that counts corresponding symbol frequencies (see [Witten et al. 87] for details).

Compressed data structure. The operations described above are done for all patches in the hierarchy. After the preprocessing, the resulting encoded multiresolution hierarchy is represented by

- The coarsest level patch $\tilde{H}_{0,0}^{(0)}$ at level 0, which covers the entire terrain;
- Refinement data (prediction errors) for all patches in the hierarchy, excepting those at the finest resolution.

An encoded representation of the quadtree shown in Figure 2.2 is illustrated in Figure 2.6. Note that nodes at level $l = l_0$ do not contain data.

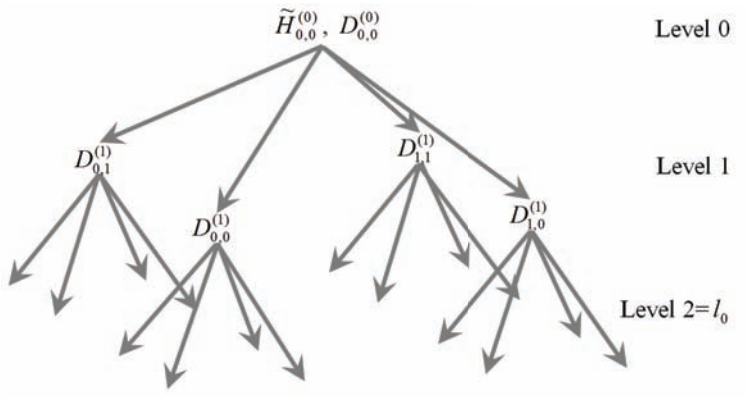


Figure 2.6. Compressed hierarchy structure.

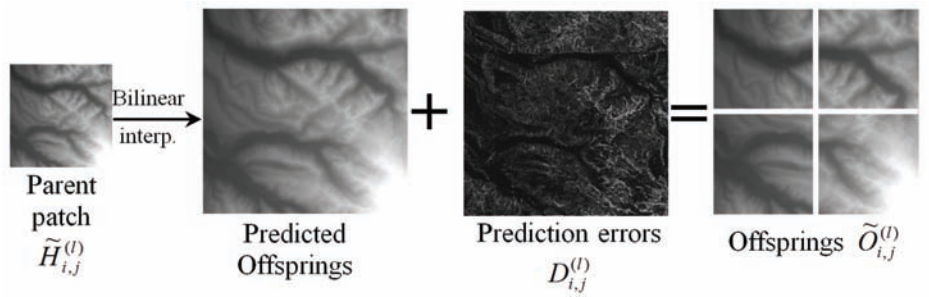


Figure 2.7. Offspring reconstruction diagram.

The matrix $D_{i,j}^{(l)}$ contains all the information that is required to reconstruct the offspring of patch $\tilde{H}_{i,j}^{(l)}$ (Figure 2.7). Repeating the decompression process starting from the root, the entire original height map can be progressively reconstructed.

In contrast to our original approach [Yusov and Shevtsov 11], in the compression scheme presented here, a single error threshold is used for each level of the hierarchy. This has the following benefits:

- Compression and decompression can be performed in one step (the original approach consists of two steps);
- A higher compression ratio;
- A simpler error control.

2.3.3 GPU-Based Decompression

In our system, the decompression is done in three steps:

1. At the first step, the CPU reads an arithmetically-encoded bit stream and uploads decoded prediction errors $D_{i,j}^{(l)}$ to the temporary $(2 \cdot 2^n + 1) \times (2 \cdot 2^n + 1)$ 8-bit texture T_D.
2. At the second step, child patch height maps $\tilde{O}_{i,j}^{(l)}$ are reconstructed by the GPU and stored as a temporary $(2 \cdot 2^n + 1) \times (2 \cdot 2^n + 1)$ 16-bit texture T_O.
3. At the third step, appropriate regions of T_O are copied to four offspring $(2^n + 1) \times (2^n + 1)$ height-map textures.

At the second step, a full-screen quad covering the entire T_O texture is rendered; the required operations are done by the pixel shader. During the rasterization, the parent patch height-map is interpolated using hardware-supported bilinear filtering; interpolation errors are loaded from T_D and added to the predicted values. The corresponding code is presented in Listing 2.1.

```
#define HEIGHT_MAP_SAMPLING_SCALE 65535.f
float DecompressOffspringsPS(VS_OUTPUT In) : SV_TARGET
{
   // Clamp parent patch height map UV coordinates to the
   // decoded area
   float2 ParentPatchUV = clamp( In.m_ParentPatchUV,
         g_ParentPatchUVClampArea.xy,
         g_ParentPatchUVClampArea.zw );
   // Filter parent patch height map and get the predicted value
   int PredictedElev = g_tex2DParentHeightMap.SampleLevel(
                                 samLinearClamp, ParentPatchUV, 0)
                                      *HEIGHT_MAP_SAMPLING_SCALE;
   // Load residual
   int iQuantizedResidual =
     g_tex2DResiduals.Load( int3(In.m_ResidualIJ.xy, 0) );
   // Quantize predicted height
  int iQuantizedPredictedElev =
           QuantizeValue( PredictedElev, g_uiReconstrPrecision );
   // Add refinement label and dequantize
   int ReconstructedChildElev = DequantizeValue(
      iQuantizedPredictedElev + iQuantizedResidual,
      g_uiReconstrPrecision );
   // Scale to [0,1]. Note that 0.f corresponds to 0u and 1.f
   // corresponds to 65535u
   return (float)ReconstructedChildElev / HEIGHT_MAP_SAMPLING_SCALE;
}
```

Listing 2.1. GPU-part of the decompression.

Important point. UNORM textures are filtered differently on Nvidia and ATI GPUs. On ATI hardware, the texture sampler returns an exact weighted result, while on Nvidia hardware, the filtering result is always in the form of f/N where f and N are integers and N is the normalization constant ($N = 65535$ for 16-bit UNORM texture). In the latter case, it is necessary to filter the texture manually to get a precise filtering result. The demo determines if texture filtering is done precisely by the GPU and uses a hardware-supported filtering unit or performs filtering manually, if it is not.

GPU-based decompression has the following benefits:

- It reduces the data-transfer overhead: instead of uploading 16-bit decompressed height maps to the GPU memory, only 8-bit prediction errors are uploaded.
- It eliminates the need to duplicate the data in main and GPU memory.
- It improves decompression performance.

Note that in practice, a number of coarse levels (three, for instance) are kept in main memory as well to perform collision detection.

2.3.4 Extending Height Maps to Guarantee Seamless Normal Maps

Normal maps required to perform lighting calculations are generated at run time when the corresponding patch is decompressed. This is done on the GPU using discrete gradient approximation. While $(2^n + 1) \times (2^n + 1)$ patches provide geometric continuity, normals located on the boundaries of neighboring patches are not calculated equally, which leads to noticeable seams. To guarantee consistent normal map calculation at the patch boundaries, the finest resolution patches contain one additional contour around the patch perimeter and, thus, have a size of $(2^n+3)\times(2^n+3)$. To encode these additional contours, parent patch boundary samples are extended as if filtering with a clamp mode were used. This is what the first operator of the `DecompressOffspringsPS()` shader does:

```
float2 ParentPatchUV = clamp( In.m_ParentPatchUV,
                              g_ParentPatchUVClampArea.xy,
                              g_ParentPatchUVClampArea.zw );
```

All patches are stored as $(2^n + 4) \times (2^n + 4)$ textures. Patches at coarser resolution levels do not contain additional contour and have an encoded area of $(2 \cdot 2^n + 1) \times (2 \cdot 2^n + 1)$. The remaining samples of the patch height maps are obtained through linear extrapolation of the boundary samples.

2.3.5 Guaranteed Patch Error Bound

During the quadtree construction, each patch in the hierarchy is assigned a world-space approximation error, which conservatively estimates the maximum geometric deviation of the patch's reconstructed height map from the underlying original full-detail height map. This value is required at runtime to estimate the screen-space error and to construct the patch-based adaptive model, which approximates the terrain with the specified screen-space error (Section 2.4).

Let's denote the upper error bound of patch $\tilde{H}_{i,j}^{(l)}$ by $\mathrm{Err}(\tilde{H}_{i,j}^{(l)})$. To calculate $\mathrm{Err}(\tilde{H}_{i,j}^{(l)})$, we first calculate the *approximation error* $\mathrm{Err}_{\mathrm{Appr}}(H_{i,j}^{(l)})$, which is the upper bound of the maximum distance from the patch's *precise* height map to the samples of the underlying full-detail (level l_0) height map. It is recursively calculated using the same method as that used in ROAM [Duchaineau et al. 97] to calculate the thickness of the nested bounding volumes (called wedges):

$$\mathrm{Err}_{\mathrm{Appr}}(H_{i,j}^{(l_0)}) = 0,$$

$$\mathrm{Err}_{\mathrm{Appr}}(H_{i,j}^{(l)}) = \mathrm{Err}_{\mathrm{Int}}(H_{i,j}^{(l)}) + \max_{s,t=\pm 1} \{\, \mathrm{Err}_{\mathrm{Appr}}(H_{2i+s,2j+t}^{(l+1)}) \,\}, l = l_0 - 1, ...0,$$

where $\mathrm{Err}_{\mathrm{Int}}(H_{i,j}^{(l)})$ is the maximum geometric deviation of the linearly interpolated patch's height map from its offspring. A 1D illustration for determining $\mathrm{Err}_{\mathrm{Int}}(H_{i,j}^{(l)})$ is given in Figure 2.8.

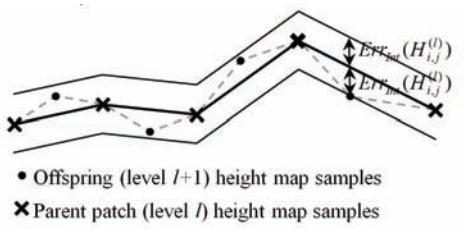


Figure 2.8. 1D illustration for the patch interpolation error. (Courtesy of WSCG.)

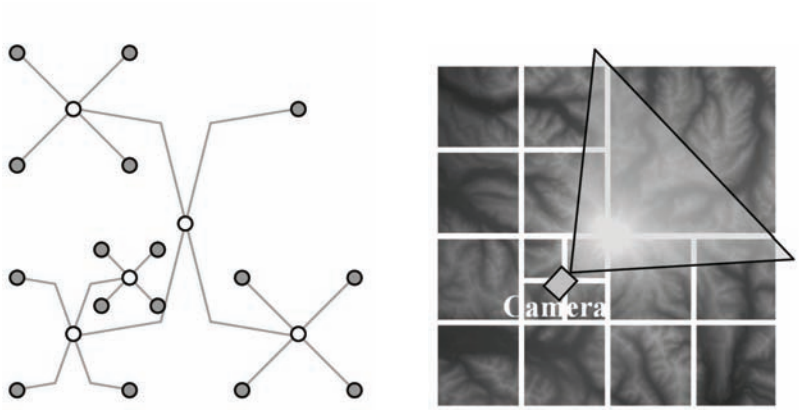
Since the reconstructed height map for the patch $\tilde{H}_{i,j}^{(l)}$ deviates from the exact height map by at most δ, the final patch's upper error bound is given by

$$\mathrm{Err}(\tilde{H}_{i,j}^{(l)}) = \mathrm{Err}_{\mathrm{Appr}}(H_{i,j}^{(l)}) + \delta.$$

2.4 Hardware-Accelerated Terrain Tessellation

At run time, an unbalanced patch quadtree (Figure 2.9(a)) is maintained in decompressed form. It defines the block-based terrain approximation (Figure 2.9(b)), in which each patch satisfies the given screen-space error tolerance for the current camera position and view parameters.

The unbalanced quadtree is cached in GPU memory and is updated as the camera moves. Since we already have the maximum geometric error $\mathrm{Err}(\tilde{H}_{i,j}^{(l)})$



(a) (b)

Figure 2.9. (a) Unbalanced quadtree maintained in decompressed form. (b) Corresponding approximation of the terrain.

for the vertices within each patch, we can conservatively estimate the maximum screen-space vertex error $\mathrm{Err}_{\mathrm{Scr}}(\tilde{H}_{i,j}^{(l)})$ for that patch using the standard LOD formula (see [Ulrich 00, Levenberg 02]):

$$\mathrm{Err}_{\mathrm{Scr}}(\tilde{H}_{i,j}^{(l)}) = \gamma \frac{\mathrm{Err}(\tilde{H}_{i,j}^{(l)})}{\rho(c, V_{i,j}^{(l)})},$$

where $\gamma = \frac{1}{2}\max(R_h \cdot \mathrm{cotan}(\varphi_h/2),\ R_v \cdot \mathrm{cotan}(\varphi_v/2))$, R_h, and R_v are horizontal and vertical resolutions of the view port, φ_h and φ_v are the horizontal and vertical camera fields of view, and $\rho(c, V_{i,j}^{(l)})$ is the distance from the camera position c to the closest point on the patch's bounding box $V_{i,j}^{(l)}$.

Screen-space error estimation $\mathrm{Err}_{\mathrm{Scr}}(\tilde{H}_{i,j}^{(l)})$ is compared to the user-defined error threshold ε for each patch in the current model. Depending on the results of the comparison, new patches are extracted from the compressed representation for these regions where additional accuracy is required; on the other hand, patches that represent the surface with the exceeded precision are destroyed and replaced with their parents.

Each patch in the current unbalanced quadtree is triangulated using the hardware tessellation unit as described below. Section 2.4.1 gives a brief overview of the new stages of the Direct3D 11 pipeline that are used in the proposed method.

2.4.1 Tessellation Stages in Direct3D 11

The Direct3D 11 pipeline contains new stages that drive tessellation entirely on the GPU (see Figure 2.10):

- *Hull Shader (programmable stage).* Transforms a set of input control points (from a vertex shader) into a set of output control points.

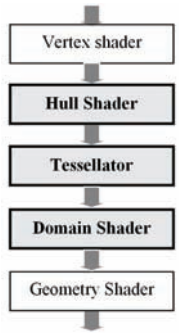


Figure 2.10. Tessellation stages in the Direct3D 11 pipeline.

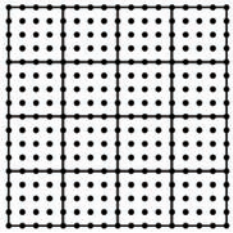


Figure 2.11. Subdividing a 17 × 17 patch into 5 ×5 ($d_0 = 2$) tessellation blocks.

- *The Tessellator (fixed stage).* Subdivides a domain (quad, tri, or line) into many smaller objects (triangles, points, or lines).
- *Domain Shader (programmable stage).* Calculates the vertex position of a subdivided point in the output patch.

2.4.2 Tessellation Blocks

To construct an adaptive view-dependent triangulation, each patch in the decompressed model is subdivided into the small equal-sized blocks ($(2^{d_0}+1)\times(2^{d_0}+1)$ in size) that we call *tessellation blocks* (Figure 2.11). For instance, a 65 × 65 patch can be subdivided into the 4 × 4 grid of 17 × 17 tessellation blocks or into the 8 × 8 grid of 9 × 9 blocks, and so on.

Each tessellation block can be rendered at the resolution 2^d, $d = 1, 2, ..., d_0$ (Figure 2.12). Optimal resolution for each block is determined by the hull shader.

To precisely determine the degree of simplification for each block, a series of block errors is calculated when the patch is created. These errors represent the maximum geometric approximation error of the terrain region covered by the tessellation block rendered at a particular resolution *d*.

Let's denote the error of the tessellation block located at the (r, s) position in the patch $H_{i,j}^{(l)}$ rendered at resolution 2^d by $\lambda_{r,s}^{(d)}$. Then, $\lambda_{r,s}^{(d)}$ can be computed as a sum of two components: the patch error bound and the maximum deviation of the patch samples to the simplified triangulation of the tessellation block:

$$\lambda_{r,s}^{(d)} = \max_{v \notin T_{r,s}^{(d)}} \rho(v, T_{r,s}^{(d)}) + \mathrm{Err}(\tilde{H}_{i,j}^{(l)}), \quad d = 1, 2, \ldots,$$

where $T_{r,s}^{(d)}$ is the tessellation block (r, s) triangulation at resolution 2^d and $\rho(v, T_{r,s}^{(d)})$ is the vertical distance from the vertex v to the triangulation $T_{r,s}^{(d)}$. Two- and four-times simplified triangulations as well as these samples (dotted circles) of the patch height map that are used to calculate $\lambda_{r,s}^{(2)}$ and $\lambda_{r,s}^{(1)}$ are shown in Figure 2.12 (center and right images, respectively).

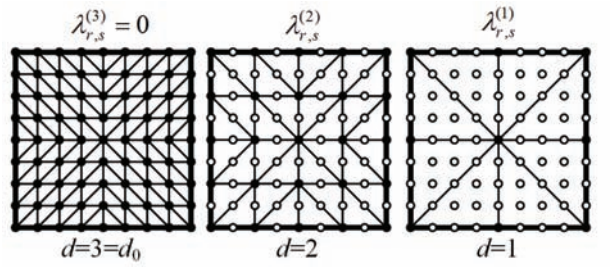


Figure 2.12. Different resolution levels of a 9×9 ($d_0 = 3$) tessellation block. (Courtesy of WSCG.)

In our implementation, we calculate errors for four simplification levels such that a tessellation block triangulation can be simplified by a maximum factor of $(2^4)^2 = 256$ (if the tessellation block is large enough). This enables us to store the tessellation block errors as a four-component vector. The corresponding code is presented in Listing 2.2.

```
float4 CalculateTessBlockErrors(float2 ElevationMapUV,
                                float fPatchApproxErrorBound,
                                float fElevDataTexArrayIndex)
{
#define MAX_ERR 1.7e+34f
  float TessBlockErrors[4] =
    { MAX_ERR, MAX_ERR, MAX_ERR, MAX_ERR };

#define GET_ELEV(Col, Row) \
   g_tex2DElevationMap.SampleLevel(samPointClamp, \
     ElevationMapUV.xy + float2(Col,Row)*g_ElevDataTexelSize, \
     ) * HEIGHT_MAP_SAMPLING_SCALE

  for(int iCoarseLevel = 1; iCoarseLevel <= 4; iCoarseLevel++)
  {
    float fTessBlockCurrLevelError = 0.f;
    int iStep = 1 << iCoarseLevel;
    // Minimum tessellation factor for the tessellation
    // block is 2, which corresponds to the step
    // g_iTessBlockSize/2. There is no point in considering
    // larger steps
    if( iStep > g_iTessBlockSize/2 )
      break;

    // Tessellation block covers height map samples in the
    // range [-g_iTessBlockSize/2, g_iTessBlockSize/2]
    for(int iRow = -(g_iTessBlockSize>>1);
            iRow <= (g_iTessBlockSize>>1); iRow++)
        for(int iCol = -(g_iTessBlockSize>>1);
                iCol <= (g_iTessBlockSize>>1); iCol++)
        {
          int iCol0 = iCol & (-iStep);
          int iRow0 = iRow & (-iStep);
          int iCol1 = iCol0 + iStep;
```

```
            int iRow1 = iRow0 + iStep;

            float fHorzWeight =
              ((float)iCol - (float)iCol0) / (float)iStep;
            float fVertWeight =
              ((float)iRow - (float)iRow0) / (float)iStep;

            float fElev00 = GET_ELEV(iCol0, iRow0);
            float fElev10 = GET_ELEV(iCol1, iRow0);
            float fElev01 = GET_ELEV(iCol0, iRow1);
            float fElev11 = GET_ELEV(iCol1, iRow1);
            float fInterpolatedElev =
              lerp( lerp(fElev00, fElev10, fHorzWeight ),
                    lerp(fElev01, fElev11, fHorzWeight ),
                    fVertWeight );

            float fCurrElev = GET_ELEV(iCol, iRow);
            float fCurrElevError =
              abs(fCurrElev - fInterpolatedElev);
            fTessBlockCurrLevelError =
              max(fTessBlockCurrLevelError, fCurrElevError);
          }

      TessBlockErrors[iCoarseLevel-1] =
          fTessBlockCurrLevelError;
    }

    return float4(TessBlockErrors[0], TessBlockErrors[1],
                  TessBlockErrors[2], TessBlockErrors[3]) +
           fPatchApproxErrorBound;
}
```

Listing 2.2. Calculating tessellation block errors.

One may ask why we don't use a compute shader to calculate tessellation block errors. The primary reason is that we do not need special features of the compute shader, such as random memory access, shared memory, or thread synchronization, to calculate the errors. Besides, we will be required to select such parameters as thread group size, which can affect performance. With a pixel shader-based approach, all these tasks are done by the driver; thus this approach is more universal and efficient. Another problem is that currently it is not possible to bind a 16-bit UNORM or UINT resource as an unordered access view.

2.4.3 Selecting Tessellation Factors for the Tessellation Blocks

When the patch is to be rendered, it is necessary to estimate how much its tessellation blocks' triangulations can be simplified without introducing unacceptable error. This is done using the current frame's world-view projection matrix. Each tessellation block is processed independently, and for each block's edge, a tessellation factor is determined. To eliminate cracks, tessellation factors for shared edges of neighboring blocks must be computed in the same way. The tessellation

factors are then passed to the tessellation stage of the graphics pipeline, which generates the final triangulation.

Tessellation factors for all edges are determined identically. Let's consider some edge and denote its center by e_c. Let's define edge errors $\Lambda_{e_c}^{(d)}$ as the maximum error of the tessellation blocks sharing this edge. For example, block (r, s) left edge's errors are calculated as follows:

$$\Lambda_{e_c}^{(d)} = \max\left(\lambda_{r-1,s}^{(d)}, \lambda_{r,s}^{(d)}\right), \quad d = 1, 2, \ldots.$$

Next let's define a series of segments in a world space specified by the end points $e_c^{(d),-}$ and $e_c^{(d),+}$ determined as follows:

$$e_c^{(d),-} = e_c - \Lambda_{e_c}^{(d)}/2 \cdot e_z,$$
$$e_c^{(d),+} = e_c + \Lambda_{e_c}^{(d)}/2 \cdot e_z,$$

where e_z is the world space z-axis unit vector. Thus $e_c^{(d),-}$ and $e_c^{(d),+}$ define a segment of length $\Lambda_{e_c}^{(d)}$ directed along the z-axis such that the edge center e_c is located in the segment's middle.

If we project this segment onto the viewing plane using the world-view projection matrix, we will get the edge screen-space error estimation (Figure 2.13) given that the neighboring tessellation blocks have detail level 2^d. We can then select the minimum tessellation factor d for the edge that does not lead to unacceptable error as follows:

$$d = \arg\min_d \operatorname{proj}(\overline{e_c^{(d),-}, e_c^{(d),+}}) < \varepsilon.$$

The code that calculates the edge tessellation factor is presented in Listing 2.3.

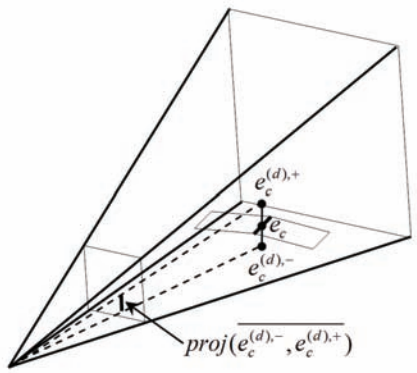


Figure 2.13. Calculating edge screen-space error. (Courtesy of WSCG.)

```
float GetScrSpaceError(float EdgeError, float3 EdgeCenter)
{
  float3 ShftdPoint1 = EdgeCenter - float3(0, 0, EdgeError/2.f);
  float3 ShftdPoint2 = EdgeCenter + float3(0, 0, EdgeError/2.f);
  float4 ShftdPoint1_PS =
    mul( float4(ShftdPoint1, 1), g_mWorldViewProj );
  float4 ShftdPoint2_PS =
    mul( float4(ShftdPoint2, 1), g_mWorldViewProj );

  ShftdPoint1_PS /= ShftdPoint1_PS.w;
  ShftdPoint1_PS /= ShftdPoint2_PS.w;
  ShftdPoint1_PS.xy *= g_ViewPortSize.xy/2;
  ShftdPoint2_PS.xy *= g_ViewPortSize.xy/2;
  return length(ShftdPoint1_PS.xy - ShftdPoint2_PS.xy);
}
#define MIN_EDGE_TESS_FACTOR 2
float CalculateEdgeTessFactor(float4 Errors,
                              float3 EdgeCenter)
{
  float EdgeTessFactor = BLOCK_SIZE;
  float4 ScrSpaceErrors;
  ScrSpaceErrors.x = GetScrSpaceError(Errors.x, EdgeCenter);
  ScrSpaceErrors.y = GetScrSpaceError(Errors.y, EdgeCenter);
  ScrSpaceErrors.z = GetScrSpaceError(Errors.z, EdgeCenter);
  ScrSpaceErrors.w = GetScrSpaceError(Errors.w, EdgeCenter);

  // Compare screen-space errors with the threshold
  float4 Cmp =
      (ScrSpaceErrors.xyzw < g_fScrSpaceErrorThreshold.xxxx);
  // Calculate number of errors less than the threshold
  float SimplPower = dot( Cmp, float4(1,1,1,1) );
  // Compute simplification factor
  float SimplFactor = exp2( SimplPower );
  // Calculate edge tessellation factor
  EdgeTessFactor /= SimplFactor;
  return max(EdgeTessFactor, MIN_EDGE_TESS_FACTOR);
}
```

Listing 2.3. Calculating tessellation factors.

The same selection process is used for each edge. The tessellation factor for the block interior is then defined as the minimum of its edge tessellation factors. This method assures that tessellation factors for shared edges of neighboring blocks are computed consistently and guarantees seamless patch triangulation. An example of tessellation factors assigned to tessellation block edges and the final patch triangulation is given in Figure 2.14.

To hide gaps between neighboring patches, we construct "vertical skirts" around the perimeter of each patch as proposed by T. Ulrich [Ulrich 00]. The top of the skirt matches the patch's edge and the skirt height is selected such that it hides all possible cracks. The skirts are rendered as additional tessellation blocks surrounding the patch, which have a special triangulation (see Figure 2.15).

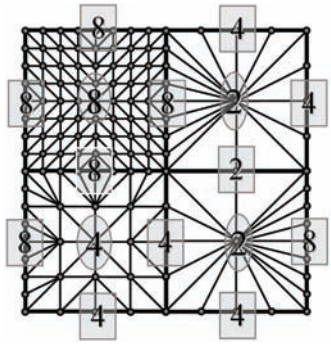


Figure 2.14. Tessellation factors assigned to block edges and the final patch triangulation.

The method can be summarized as follows:

- `D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCH_LIST` primitive topology with fractional even partitioning is set.
- The vertex shader simply passes the data to the hull shader.
- The hull shader constant function calculates the tessellation factor for each edge (Figure 2.16 (left)) and passes the data to the tessellator.
- The tessellator generates topology and domain coordinates (Figure 2.16 (middle)) that are passed to the domain shader.
- The domain shader fetches the height-map value from the appropriate texture and calculates world-space position for each vertex (Figure 2.16 (right)).
- The resulting triangles then pass in a conventional way via the rasterizer.

Since our method enables using a small screen-space error threshold (two pixels or less) with high frame rates (more than 300 fps in a typical setup), we

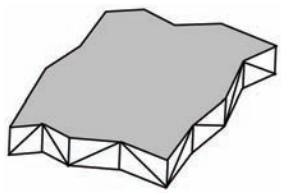


Figure 2.15. Vertical skirts around the patch perimeter.

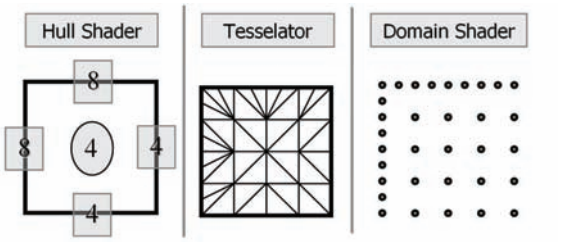


Figure 2.16. Tessellation stages.

did not observe any popping artifacts during our experiments even though there is no morph between successive LODs in our current implementation. In addition, on large thresholds, we noticed that it is much more important to perform a morph of the diffuse texture and the normal map rather than a geometry morph; these are implemented in the demo.

The proposed technique has the following benefits:

- Triangulation is fully constructed by the hardware, which saves CPU time and eliminates data transfer necessary to upload adaptive triangulation topology.
- The triangulation is computed using the current frame world-view projection matrix, it is adaptive to each camera position, and it tolerates the user-defined screen-space error bound.
- After updating the tessellation block errors, the triangulation is updated automatically.

2.5 Texturing

Terrain texture is generated procedurally based on the local terrain height and slope. The proposed method is based on the technique presented in [Dachsbacher and Stamminger 06]. The texturing method is driven by the list of materials. Each material is characterized by the unique texture that is tiled across the surface and the allowable ranges of height and slope. All material textures are stored in single texture array objects that enable selecting materials in the pixel shader.

Each patch has its unique 8-bit material index map that stores an index of the material that is the most appropriate for each patch region. In contrast to a traditional alpha texture that stores weights for four materials in its r, g, b, and a channels, the material index map stores only one 8-bit index. The material index map is calculated on the GPU by selecting the material with the best matching height and slope.

Terrain texturing is performed in the pixel shader. The shader reads the material index map and samples the appropriate texture-array element. To implement smooth transitions between materials, the shader reads the four nearest materials and blends them with appropriate weights. The shader code snippet is presented in Listing 2.4.

```
float2 TexUVSclaed =
  In.DiffuseTexUV.xy*g_MtrlIdxTexSize.xy - float2(0.5, 0.5);
float2 TexIJ = floor(TexUVSclaed);
float2 BilinearWeights = TexUVSclaed - TexIJ;
TexIJ = (TexIJ + float2(0.5, 0.5)) / g_MtrlIdxTexSize.xy;

float2 Noise = g_tex2DNoiseMap.Sample(samLinearWrap,
                        In.DiffuseTexUV.xy * 50).xy;
BilinearWeights =
  saturate(BilinearWeights + 0.5*(Noise - 0.5));

float4 Colors[2][2];
for(int i=0; i<2; i++)
 for(int j=0; j<2; j++)
 {
   float MtrIdx = g_tex2DMtrlIdx.Sample( samPointClamp,
                    TexIJ.xy, int2(i,j) )*255;
   Colors[i][j] = g_tex2DSrfMtrlArr.Sample(samAnisotropicWrap,
                    float3(In.TileTexUV.xy, MtrIdx) );
 }

SurfaceColor = lerp(
  lerp(Colors[0][0], Colors[1][0], BilinearWeights.x),
  lerp(Colors[0][1], Colors[1][1], BilinearWeights.x),
  BilinearWeights.y );
```

Listing 2.4. Calculating procedural surface color in a pixel shader.

The advantages of the proposed texturing technique are the following:

- There is a high-detailed surface. Material textures are tiled across the terrain and provide high-quality details.
- The number of materials is practically unlimited; the performance does not depend on the number of materials used.
- Each patch has an 8-bit material index texture instead of a 32-bit alpha texture storing blend weights in r, g, b, and a channels.
- After the material index map is regenerated, the terrain texture is updated automatically.

2.6 Dynamic Modifications

2.6.1 Asynchronous Tasks

All time-consuming tasks in the system are done asynchronously. Each node of the current unbalanced quadtree can be assigned one of the following asynchronous tasks:

- *Increase LOD*. This task is assigned to the quadtree node when its level of detail is about to increase.
- *Decrease LOD*. This task is assigned to the quadtree node when its level of detail is about to decrease.
- *Recompress*. This task is assigned to the coarse-level node when its offspring need to be recompressed.

The tasks are scheduled for execution and asynchronously processed by the system. Before any new task can be assigned to the node, the previous one must be completed.

2.6.2 Modifications

Terrain modifications are represented by the displacement map (which can be negative or positive). The modification object contains texture storing the displacement values.

Modifying the terrain consists of two parts. The first part is instant modification that is applied to each patch in the current unbalanced quadtree affected by the modification. Instant modification is performed by rendering to the appropriate regions of the affected patches. Since rendering to texture is done very efficiently, instant modifications are very inexpensive and cause almost no performance drop. When a patch is modified, its tessellation block errors, normal map, and material-index textures are marked as invalid. These textures are regenerated just before the patch is rendered. Since the triangulation topology is constructed entirely on the GPU, it will be updated automatically as the new tessellation block errors are calculated.

The second part is an asynchronous task that re-encodes the modified height map into the compressed representation. There are two possible cases that are processed differently:

1. The modified patch is located in the finest resolution level.
2. The modified patch is located in the coarse level.

Modifying patches in the finest resolution level. In this case, affected patches simply accumulate modifications and are marked as "modified." No data compression is performed at this stage. The height map is compressed asynchronously by the "Decrease LOD" task when affected patches are about to be destroyed. The "Decrease LOD" task checks if patches are modified and performs data recompression if necessary. Since the height map is kept in GPU memory only, it is necessary to read the data back to the main memory. This is done using a number of staging resources. The modified height map is copied to the unused staging resource, which is then mapped with the flag `D3D11_MAP_FLAG_DO_NOT_WAIT` in order to not stall the system. If data is not ready, the attempt to map the texture is repeated on the next frame. After the data is read back, the steps described in Section 2.3.2 are repeated and new prediction errors are stored in the data base. The parent patch is then marked as "modified" as well. It will be re-encoded in the same manner just before it is destroyed.

Modifying patches in coarse resolution levels. The second case is more complex since we cannot apply modification to the coarse patch only, because we will lose some data. In fact, if patch is located at a very coarse level, the whole modification can be lost. Thus, to correctly apply modification, we need to perform the following steps:

1. Decompress patches at the finest resolution level affected by the modification;
2. Apply modification;
3. Re-encode the modified regions in bottom-up order and update the data base.

All these tasks are done on the CPU in a separate thread asynchronously to the rendering thread. As in the previous case, the data is first read back to the CPU memory before starting data recompression.

It is possible that a coarse level patch is modified before the pior recompressing task is finished. In this case, all modifications are added to the list attached to the modified quadtree node. After the recompression task is completed, a new task is started that applies all modifications in the list at the same time.

2.7 Implementation Details

Two different GPU resource management strategies are implemented in the demo app. The first one is the conventional GPU resource cache. The cache stores unused textures released when patches are destroyed. When a new patch is created, appropriate resources are extracted from the cache. This eliminates expensive resource creation.

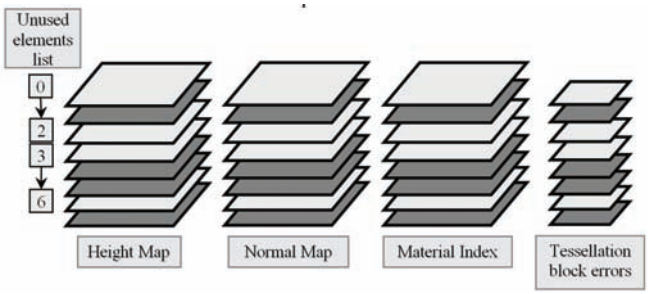


Figure 2.17. Texture arrays storing patch data.

The second strategy exploits the texture-array object introduced in DX10. The texture array contains a collection of identical textures that are interpreted by DirectX as a single object. The texture array can be indexed in the shader such that each individual array element can be accessed. Implementation with the texture arrays has the following advantages that improve performance:

- Texture arrays function as a GPU resource cache.
- The whole terrain can be rendered using a single instanced draw call.
- Multiple textures (normal map, tessellation block errors, and material-index map) can be updated in a single draw call.

The following four texture arrays are naturally maintained in our system (see Figure 2.17): height map, normal map, material index, and tessellation block errors. As it was noted earlier, height, normal, and material index maps are $(2^n+4)\times(2^n+4)$ in size with additional samples required to guarantee consistent surface shading at patch boundaries.

To implement resource cache functionality, a list of unused subresources is supported. When a patch is created, an unused subresource is found in the list. When the patch is destroyed, corresponding texture array elements are marked as free and become available for new patches.

Since all patches in the model share the same topology `D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCH_LIST`, the whole terrain can be rendered using a single instanced draw call. The per-instance data buffer is populated with the patch location, level in the hierarchy, and the texture index. Patch-rendering shaders then access appropriate texture array elements using the index and fetch the required data.

As discussed in previous sections, tessellation block errors, normal maps, and material indices are computed on the GPU for each new or modified patch. Texture arrays enable updating multiple textures during one draw call. For

this purpose, a list of texture indices to update is uploaded to the GPU memory and a single instanced draw call is invoked. The geometry shader selects the texture array element on which rendering should be performed using the `SV_RenderTargetArrayIndex` semantic.

As shown in Section 2.8, instanced rendering improves performance by a factor of more than two.

2.8 Performance

For our tests we used the Puget Sound elevation data set,[1] which is a common benchmark for terrain-rendering systems. The height-map size is 16385 × 16385 at 10 m spacing, which results in a terrain size of 160 × 160 km. The patch and tessellation block sizes in our experiments were chosen to be 129 × 129 and 17 ×17, respectively. The initial data set (512 MB) was compressed to 48.4 MB with 0.5 m error tolerance and 2.9 MB of auxiliary data (tessellation block errors, min/max elevations, etc.). Performance was measured on the following machine: single Intel Core i7 @2.67; 6.0 GB RAM; Nvidia GeForce GTX480.

Average performance during the recorded fly mode as well as minimal fps (in brackets) are presented in Table 2.1 (with procedural texturing disabled) and in Table 2.2 (with procedural texturing enabled).

Results presented in Table 2.1 demonstrate the efficiency of the proposed GPU-accelerated decompression and hardware-supported adaptive tessellation. Even at 1920×1200 resolution with 1 pixel tolerance, the performance never dropped below 244 fps with an average at 414 fps. With 2 pixel error threshold the visual quality is almost similar, but average frame rate increases to 818 fps. At lower quality settings the performance is even higher.

Tolerance (pixels)	**Screen Resolution**		
	1920×1200	1600×1200	1280×1024
1	411 (244)	477 (275)	612 (349)
2	818 (543)	944 (605)	1169 (776)
3	1130 (774)	1261 (858)	1462 (1080)

Table 2.1. Average (minimal) frame rates (procedural texturing disabled).

Table 2.2 shows that even with the high-quality procedural texturing, the performance is still very high (243 fps on average at 1920 × 1200 with 1 pixel screen-space error tolerance).

Table 2.3 shows that using texture arrays improves performance by a factor of more than 2. Table 2.4 reveals how performance depends on the tessellation block size for different error thresholds.

[1] Puget Sound elevation data set is available at http://www.cc.gatech.edu/projects/large_models/ps.html

Tolerance (pixels)	Screen Resolution		
	1920×1200	1600×1200	1280×1024
1	243 (177)	287 (208)	390 (270)
2	375 (276)	444 (324)	609 (449)
3	443 (313)	524 (370)	721 (513)

Table 2.2. Average (minimal) frame rates (procedural texturing enabled).

Tolerance (pixels)	Screen Resolution		
	1920×1200	1600×1200	1280×1024
1	411 / 172 (2.39 x)	477 / 201 (2.37 x)	612 / 248 (2.47 x)
2	818 / 372 (2.2 x)	944 / 425 (2.22 x)	1169 / 506 (2.31 x)

Table 2.3. Comparison of average frame rates for rendering with texture arrays and conventional GPU resource cache (procedural texturing disabled).

As can be seen from Table 2.4, the optimal tessellation block size is 17 × 17. It provides the best trade-off between adaptability and the maximum possible simplification ratio. With a larger tessellation block size (33 × 33), the triangulation became less adaptive since a single singularity may result in a maximum triangulation density for the whole block. With smaller sizes (9 × 9 and 5 × 5), the maximum triangulation simplification ratio is limited to 16 and 4, respectively, which are most often insufficient.

Dynamic terrain modifications affect performance insignificantly. For instance, with procedural texturing enabled and constantly modifying terrain, the performance dropped by less than 6% from 375 to 354 fps.

For a typical flyover (1920 × 1200, 2 pixel threshold), no more than 512 patches were required for rendering. Thus, expected GPU memory requirements are the following: 132 × 132 × (2 bytes for height + 2 bytes for normal + 1 byte for material index) × 512 + 16 ×16 × (2 bytes for tess block errors) × 512 = 42.8 MB.

Tolerance (pixels)	Tessellation block size			
	5×5	9×9	**17 ×17**	33×33
1	200 (103)	**415 (242)**	**411 (244)**	256 (139)
2	376 (198)	748 (445)	**818 (543)**	575 (350)
3	529 (269)	969 (587)	**1130 (774)**	847 (558)

Table 2.4. Comparison of average (minimal) frame rates for different tessellation block sizes (procedural texturing disabled).

2.9 Conclusion

The technique presented in this paper has a number of benefits. It exploits height-map compression to substantially reduce the storage requirements and uses a hardware-supported tessellation unit to construct adaptive triangulation. The decompression is accelerated by the GPU, which improves performance. Procedural terrain texturing provides a high-quality surface texture with minimal memory requirements. The technique supports dynamic terrain modifications making the terrain fully deformable. It exploits a number of technical tricks such as the use of a texture array and instanced draw calls to improve performance. As a result, it provides high frame rates even in HD resolution, which is not negatively affected by the real-time deformations.

2.10 Acknowledgments

The author would like to thank Andrey Aristarkhov and Artem Brizitsky whose valuable input helped implement the presented technique.

Bibliography

[Dachsbacher and Stamminger 06] Carsten Dachsbacher and Marc Stamminger. "Cached Procedural Textures for Terrain Rendering" In *Shader X4 Advanced Rendering Techniques*, edited by W. Engel, pp. 457–466. Hingham, MA: Charles River Media, 2006.

[Duchaineau et al. 97] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. "ROAMing terrain: Real-Time Optimally Adapting Meshes." In *Proceedings of IEEE Visualization*, pp. 81–88. Los Alamitos, CA: IEEE Computer Society, 1997.

[Gobbetti et al. 06] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. "C-BDAM–Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering." *Computer Graphics Forum* 25:3 (2006), 333–342.

[Levenberg 02] J. Levenberg. "Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry." In *Proceedings of IEEE Visualization*, pp. 259–265. Los Alamitos, CA: IEEE Computer Society, 2002.

[Losasso and Hoppe 04] Frank Losasso and Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids." In *Proc. ACM SIGGRAPH*, pp. 769–776. New York: ACM, 2004.

[Ulrich 00] Thatcher Ulrich. "Rendering Massive Terrains Using Chunked Level of Detail." In ACM SIGGRAPH Course *Super-Size It! Scaling up to Massive Virtual Worlds*. New York: ACM, 2000.

[Witten et al. 87] Ian Witten, Radford Neal, and J. Cleary. "Arithmetic Coding for Data Compression." *Comm. ACM* 30:6 (1987), 520–540.

[Yusov and Shevtsov 11] Egor Yusov and Maxim Shevtsov. "High-Performance Terrain Rendering Using Hardware Tessellation." *Journal of WSCG* 19:3 (2011), 85–82.