SUPPLEMENTARY MATERIAL

Implementation Details of the Sample Application Using the Proposed Hybrid Terrain Representation

Çetin Koca and Uğur Güdükbay*

Department of Computer Engineering
Bilkent University
Bilkent, 06800, Ankara, Turkey

1. Introduction

We have developed a sample application for:

- creating and editing terrains,
- analyzing the memory usage of the proposed terrain representation in typical use cases,
- analyzing the performance of the proposed algorithms, and
- evaluating the visual quality of real-time terrain rendering obtained by using the proposed rendering pipeline.

The application is implemented in C++ using OpenGL (Silicon Graphics, Inc. 2012b), OpenGL Shading Language (GLSL) (Silicon Graphics, Inc. 2012a), OpenGL Extension Wrangler Library (GLEW) (Ikits and Magallon 2012), The Developer's Image Library (DevIL) (Woods *et al.* 2012), and FreeType (Turner *et al.* 2012). We provide the implementation details and algorithms used in the sample application, as well as a discussion of how the proposed approach performs compared to heightmap- and voxel-based approaches, in the following sections.

2. Sample Application

The user can generate and edit terrains in real time:

(1) Generate a coarse volumetric representation of the terrain outlining all volumetric features.
(2) Edit the terrain surface and apply fine details at an increased resolution over the coarse model by modifying heightmaps.

An overview of the terrain geometry generation process, which is a combination of the surface extraction and surface generation phases, is depicted in Figure 1.

*Corresponding author. Email: gudukbay@cs.bilkent.edu.tr

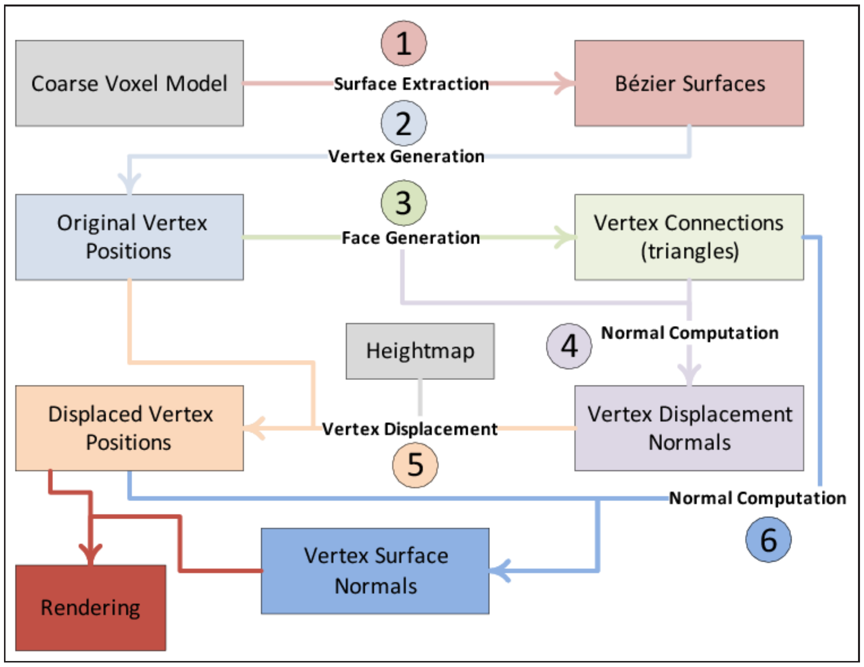

Figure 1. Overview of the terrain geometry generation.

2.1. *Terrain Editor*

2.1.1. Generating the Coarse Terrain Model

The coarse model is represented by voxels, which are stored in an octree. The model can be edited manually by selecting a specific voxel at a specific level of the octree and setting whether the voxel is occupied or not. It can also be automatically generated, entirely or partly, from a heightmap. After the heightmap is converted to the voxel model, the model can then be edited manually to create volumetric terrain features.

To generate the coarse model from the heightmap, the heightmap is first divided into blocks, depending on the desired resolution. Then the average height of each block is computed. A set of voxels lined up along the y-axis corresponds to each heightmap block. Those below the average height are set as occupied while the others remain empty. This voxel model can then be carved manually to generate volumetric terrain features, such as a cave.

The voxel editor lets the user to select the level of octree at which she or he wants to operate. Then the position of the cursor is used to compute a ray through the scene; the closest voxel at the selected level that intersects the ray is highlighted. That voxel can then be set as occupied or empty (see Figure 2). Changing the occupation status of a voxel at lower levels to occupied automatically increases the level of the octree at that branch by creating a child at each higher level. Changing the status of a voxel at a higher level removes its subtrees because that space can now be entirely represented by a single voxel.

2.1.2. Editing the Terrain Surface

A user can edit the terrain surface by deforming the heightmaps. Each voxel is associated with a certain number of patches, depending on the voxel configurations. Once a voxel is selected, the associated patches can be edited by editing the associated heightmap. A heightmap is always displayed as a 2D square texture on the user in-

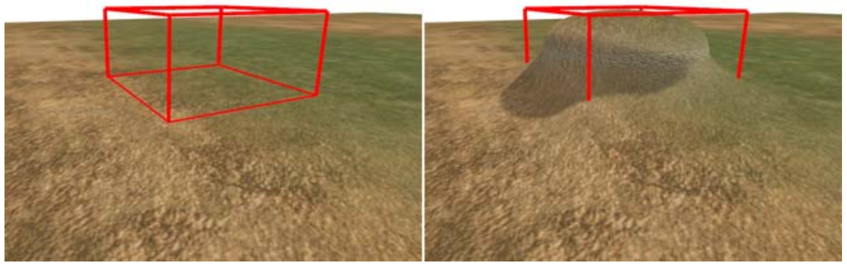

Figure 2. Manual editing of a voxel model, where the highlighted voxel is shown as a red box. Left: the highlighted voxel is empty. Right: the highlighted voxel is occupied.

terface to allow simple editing operations. The patch on which the heightmap is applied, on the other hand, can have different shapes and orientations. The mapping between the heightmap and the patch can sometimes be unintuitive; to help the user understand which part of the heightmap corresponds to which patch, the edges of the heightmap and the wireframe skeleton rendered on the selected patch are colored accordingly (see Figure 3).

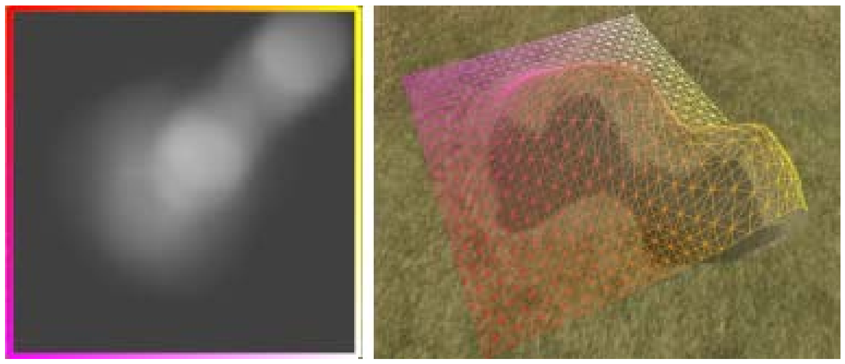

Figure 3. The terrain surface can be edited by modifying the heightmap applied to it. Left: the heightmap applied. Right: the patch being edited.

The heightmap can be edited using different types of brushes, and changes are immediately reflected on the patch.

An *extrude brush* elevates the terrain. It has the following three parameters:

- *Radius* defines the effective radius of the brush, such that the heightmap pixels outside it will not be affected.
- *Strength* determines how the brush effect will diminish as it is applied to pixels farther away from the application position.
- *Displacement factor* determines the magnitude of the displacement applied by the brush. If the displacement factor is positive, the vertices are displaced along the direction of the displacement normals; otherwise, they are displaced in the opposite direction.

It is usually preferable that the effectiveness of the brush drops as it is applied to pixels farther from the center of the application region. We use a sigmoid function (Equation 1) to compute brush effectiveness at a given distance; r is the radius of the brush, s is the strength, and d is the distance from the pixel to the brush application position. The constant δ (equal to 20 in the application) is used to convert the distance between normalized pixel coordinates on the image space into a distance comparable to the brush radius. Figure 4 demonstrates the effectiveness functions for two brushes with different strengths.

$$\begin{aligned} tanh(x) &= \tfrac{e^{2x}-1}{e^{2x}+1} \\ f(d) &= tanh\big((r - d \times \delta) \times s\big) \end{aligned} \tag{1}$$

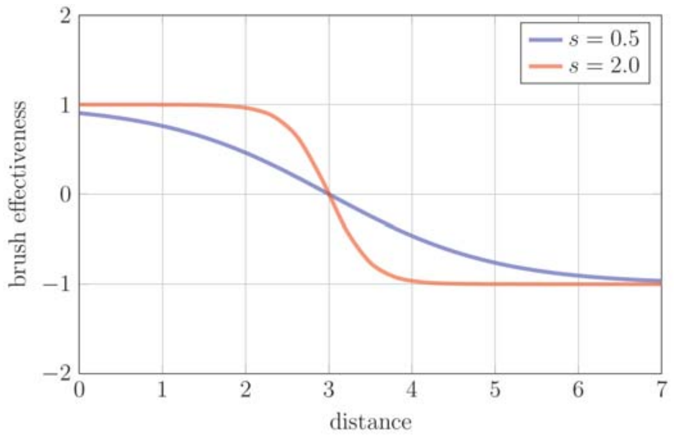

Figure 4. Sigmoid function used to compute the effectiveness of a brush at a particular distance. Blue: a brush with radius $r = 3$ and strength $s = 0.5$. Red: a brush with radius $r = 3$ and strength $s = 2$.

A *smoothing brush* smooths the terrain, and it also has three parameters: *radius*, *strength*, and *smoothing factor*. The radius and strength are the same as for the extrude brush. The smoothing factor determines how aggressively the smoothing operation is to be performed; if the factor is large, patch roughness is quickly smoothed out.

A *noise brush* applies noise. We use 3D Simplex noise Gustavson (2005), where the input to the noise function is the actual world coordinates of the vertices that correspond to the edited pixels of the heightmap. Because these coordinates are continuous the continuity of the 3D surface is guaranteed.

2.2. *Rendering Pipeline*

The rendering pipeline (see Figure 5) is implemented by programming the GPU using custom vertex and fragment shaders. It has four major stages:

(1) Vertex buffer updates (green in the figure),
(2) Index buffer updates (red in the figure),
(3) Generating shadow maps (blue in the figure), and
(4) Terrain surface rendering (orange in the figure).

2.2.1. Vertex Buffer Updates

The proposed approach uses vertex buffers to store vertex attributes such as vertex position, vertex normal, and geometry morphing parameters. The buffers are stored in the video memory because:

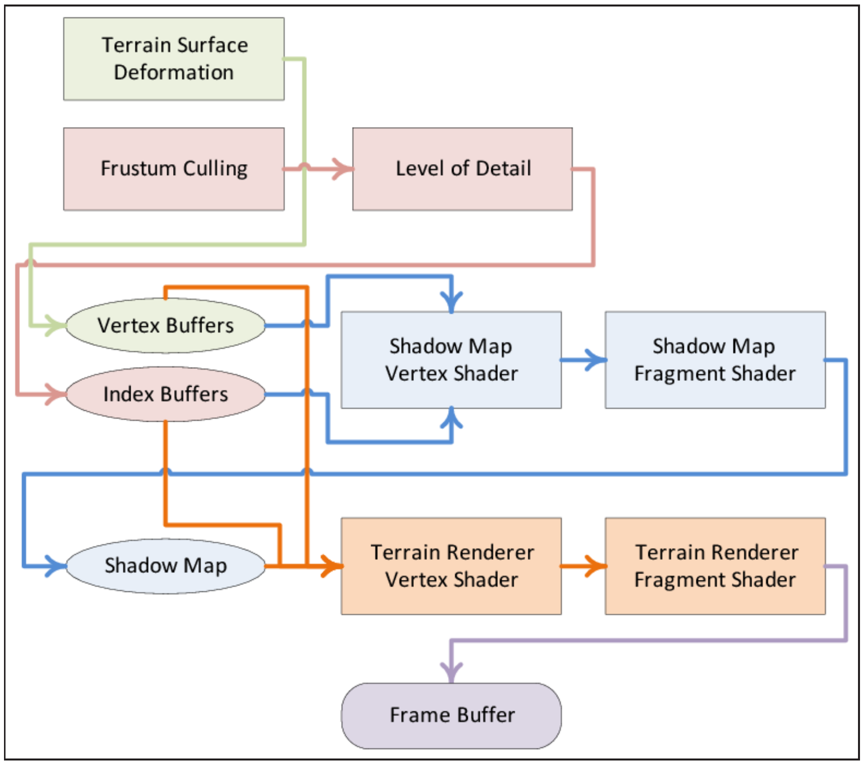
Terrain Surface Deformation
Frustum Culling
Level of Detail
Vertex Buffers
Index Buffers
Shadow Map Vertex Shader
Shadow Map Fragment Shader
Shadow Map
Terrain Renderer Vertex Shader
Terrain Renderer Fragment Shader
Frame Buffer

Figure 5. Rendering pipeline used in the sample application.

- GPU access to data in the vertex buffers is extremely fast and
- data stored in the main memory does not need to be transferred to the GPU in each frame, thus avoiding a CPU-to-GPU data-transfer bottleneck.

The vertex buffers are populated once the terrain surface is generated and ready for rendering. As long as the terrain surface is not deformed, vertex buffers do not need to be updated. When the terrain surface is deformed, the vertex attributes affected by the deformation must be updated with the data in the main memory.

Static Surface Culling. The surface extraction algorithm generates patches considering each voxel intersection volume. Because the algorithm is designed to generate connected and continuous surfaces, it can generate patches that will never be seen in practice. When a patch is generated, its control points are checked against the axis-aligned bounding box of the terrain model to determine whether the patch is redundant, i.e., when all control points are either on the bottom face or one of the side faces of the bounding box. If so, the patch is discarded. As a result of this process, although the extracted surface may no longer be connected, the visible part of the surface is still continuous. Our experiments have shown that about 40% to 55% of generated patches can be discarded. Figure 6 shows the result of static surface culling where the number of generated patches is 8,962 without culling and 3,948 with culling.

2.2.2. Index Buffer Updates

Index buffers are also stored in the video memory but they contain vertex connectivity information, that is, how the vertices should be connected to form triangles. The index buffer is the actual list of primitives to be rendered by the GPU; thus, the number of primitives represented by the index buffer determines the rendering performance. For this reason, the index buffer is updated at each frame, when the frustum culling and

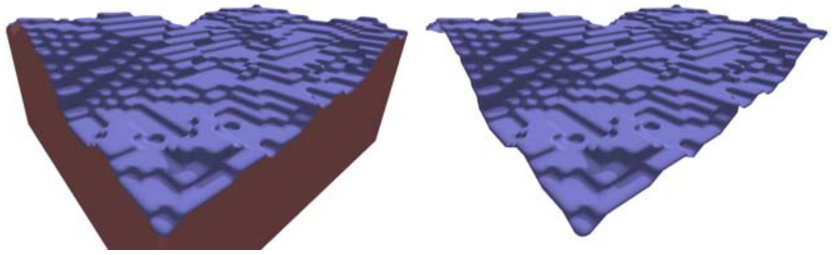

Figure 6. The result of static surface culling. Redundant surfaces are tinted with brown. Left: without static surface culling. Right: with static surface culling.

LOD functions are executed.

The algorithm that updates the index buffer operates on the octree. Each node of the octree is first checked to see whether its terrain surface bounding box is completely outside the frustum. If it is, then the subtree of that node is eliminated from further processing because it does not contribute to the rendering process. The patch LODs associated with the voxels that pass the frustum test are then computed. The corresponding triangle vertex index data of these patches are copied to the index buffers.

Frustum Culling. It is not possible to use voxels in frustum checks because the displaced terrain surfaces may overflow the voxel limits. For frustum culling, a hierarchy of surface bounding boxes is computed and then stored in the octree in a bottom-up fashion. The bounding boxes for the leaf nodes are computed by using the patches assigned to them. Then, the bounding box of each higher-level node is computed recursively by taking the union of the child node bounding boxes. When a patch is deformed, the bounding boxes of all octree nodes on the path from the leaf node that contains the deformed patch to the root are updated. This is not an expensive operation because the octree depth is not very high.

Frustum culling must be repeated whenever the view frustum changes. When four cascaded shadow maps are used, frustum culling is done four times for rendering the shadow maps and again for rendering the actual terrain surface at each frame. Frustum culling helps improve rendering performance even when it is repeated multiple times per frame.

2.2.3. Generating Shadow Maps

The terrain surface geometry is rendered once for each cascaded shadow map. The vertex and fragment shaders used to render shadow maps are relatively simple: The former computes the distance from the vertex to the observer and applies geometry morphing to the vertex. It then multiplies the computed world coordinate with the model-view-projection matrix of the light to compute the coordinates of the projected fragment. The latter then takes the z-coordinate of the fragment, adds an offset to it to prevent z-fighting, and writes the computed value to the shadow map.

2.2.4. Terrain Surface Rendering

Once the shadow maps are generated, the actual rendering can be done using the terrain surface geometry, shadow maps, and other textures. The vertex shader performs the following tasks:

- Normalizes vertex normals. Doing this on the GPU improves performance when the terrain surface is constantly being updated (e.g., during terrain editing).
- Computes the distance from the vertex to the observer.

- Computes the LOD transition position of the vertex using the distance to the observer.
- Computes the LOD transition normal of the vertex using the distance to the observer.
- Projects the world coordinate of the vertex into the eye space by multiplying it by the view-projection matrix of the perspective camera.

The fragment shader performs the following tasks:

- Normalizes the interpolated normals because interpolating between two normalized normals does not always result in a normalized normal.
- Computes tri-planar texture coordinates.
- Computes Simplex noise values for use in multi-texturing.
- Applies multi-texturing and texture splatting.
- Performs per-pixel directional lighting computations.
- Performs per-pixel point light computations for each point light in the scene.
- Samples shadow maps several times and blends them using Gaussian filtering.
- Blends the color values computed in the texturing, lighting, and shadowing phases to compute the final pixel color.

3. Algorithms and Data Structures

3.1. *Data Structures*

Our approach uses an octree to store the volumetric representation to develop a compact representation by exploiting large groups of occupied and empty voxels. The root node encloses the entire terrain. Each octree node can be divided into eight equal-sized axis-aligned child nodes if extra LOD subspace is needed (see Figure 7 (a)).

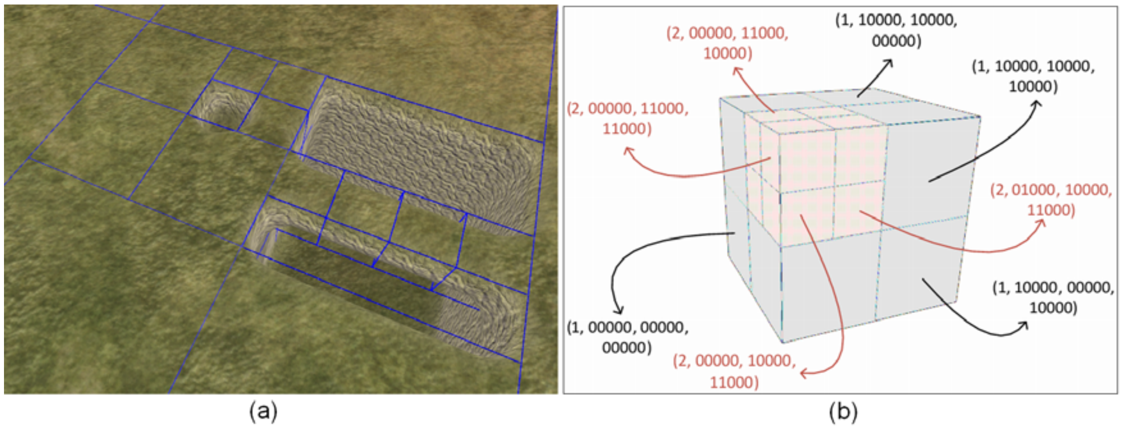

Figure 7. (a) Adaptive octree representation can increase resolution where needed. (b) Voxel indices for the adaptive octree representation (voxel level, x-index, y-index, z-index) at levels 1 and 2. Index fields are binary.

One of the most important advantages of an octree is its efficiency in running different queries on the geometry in an hierarchical manner. This feature is used by terrain editors and we use it for voxel selection and manipulation in our terrain editor. It can also significantly speed up collision queries, culling queries, and LOD queries, especially in real-time applications.

3.1.1. Voxel Structure

Each voxel has an index associated with it, which is stored in memory with the voxel (see Figure 7 (b)). Voxel size depends only on the level at which the voxel resides because the octree volume is divided by 2 at each axis at each level increment. The position of a voxel is defined as the centre of the voxel. At each level, a voxel's position is displaced along each axis by an amount equal to half the voxel size at that level in that axis, depending on whether the value of the index field for that level is 0 or 1; displacement is applied through the negative or positive side of the corresponding axis, respectively.

A voxel index is represented using four bytes in memory:

(voxel level,	x-index,	y-index,	z-index)
(four bits,	nine bits,	nine bits,	nine bits)

The voxel index structure uses a total of 31 bits of the four bytes. The final bit is used to store the information of whether the voxel is occupied or empty. This bit is not used if the index points to a voxel. A voxel index stored in this format allows up to nine additional levels other than the root level because the index fields are stored in nine bits. Consequently, the maximum height of an octree that uses this representation cannot exceed 10. An octree of height 10 has over one billion leaf nodes. In practice, an octree of height 5 or 6 provides enough resolution for most terrains. The voxel level indicates the distance from a voxel to the root of the octree.

The x-, y- and z-index fields store the index of the voxel on the x-, y- and z-axes, respectively. The i-th bit of these fields determines whether the voxel is the first or the second child of the parent voxel in the $(i-1)$-st level of the octree on the corresponding axis. The most significant bits of each field are considered: the first bits of the fields represent the child selection at *level* 0, the second bits represent the child selection at *level* 1, and so on. The number of meaningful bits in these fields is determined by the voxel level. If the voxel level is 1, then only the first bits of each field are meaningful because the child selection is done only on *level* 0. Figure 8 illustrates the computation of the x- and y-coordinates of a voxels position at level 3 from its voxel index.

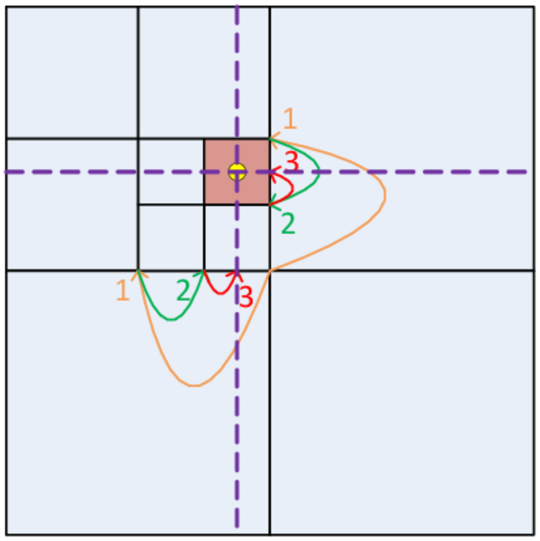

Figure 8. Computing the x- and y-coordinates of a voxel's position at level 3 from its voxel index (an xy-cross section of the voxel space is depicted).

Our voxel index representation has some advantages over traditional memory pointers:

- It takes up as much space as a memory pointer but stores additional information: the level of the voxel in the octree.
- Given a voxel index, the index of the parent voxel can be computed just by decrementing the value of the voxel level by 1.
- Given a voxel index, the index of child voxels can be computed by incrementing the voxel level by 1 and setting the i-th bit of each index field to 0 or 1, where i is equal to the incremented voxel level.
- Given a voxel index, the index of neighbor voxels at the same level can be computed by incrementing, decrementing, or keeping same the values of each index field. There are three possible operations (increment, decrement, and keep value) that can be performed on three index fields to compute the voxel index of 26 neighbor voxels.
- It simplifies the implementation of algorithms that work on the octree; it can be used to iterate voxels and move to neighbor voxels; a memory pointer can only be used to access data.
- It also saves memory space because voxel size and position can be computed given the voxel index, preventing the need to store this information.

3.1.2. Patch Structure

We choose Bézier surfaces as the underlying patch representation. Bézier curves (and surfaces) have several useful properties, such as endpoint interpolation, invariance under affine transformations and translations, lying within the convex hull of their control points, and computational efficiency when subdividing compared to other representations, such as non-uniform rational basis splines. It is possible to obtain first-order (C^1) continuity on the boundaries of two neighboring Bézier curves (thus patches) by aligning the last two control points of the first curve with the first two control points of the second curve.

Each patch has the following attributes stored in the main memory:

- A number of control points that define the surface of the patch.
- A vertex buffer that stores the vertices approximating the surface of the patch.
- An index buffer that stores the vertex indices in a particular order so as to generate the triangle list for rendering the patch.
- A heightmap associated with the patch to be used as a displacement map.
- Pointers to linked lists for vertices that are shared with other patches.
- Up to four pointers to neighbor patches with coinciding edges.

3.1.3. Vertex Structure

Vertex representation in the main memory stores the following attributes:

- Original vertex position as a 3D floating point vector (12 bytes).
- Displacement normal as a 3D floating point vector (12 bytes). This defines the direction of displacement.
- Displaced vertex position as 3D floating point vector (12 bytes). This is computed by moving the vertex in the direction of the displacement normal by an amount determined by the corresponding heightmap value.
- Actual vertex normal as a 3D floating point vector (12 bytes). This is the vertex normal computed after all displacement operations are performed. It is used for texturing and lighting computations.

- Color of the vertex as a 4D floating point vector (16 bytes). Color can be applied as a post-processing effect on the color obtained from textures. It can also be used to highlight some parts of the terrain.
- LOD transition distance of the vertex as a single floating point number (four bytes). It determines the distance at which the LOD management algorithm will move the vertex into a lower LOD position.
- Pointers to two neighbor vertices at a lower LOD (eight bytes). These vertices are accessed to compute the lower LOD transition position and normal of the current vertex when the vertex data is to be sent to the video memory.
- Index of the vertex in the GPU vertex buffers as an unsigned integer (four bytes). This is a pointer to the vertex data stored in the video memory; it is used to update that data whenever the data in the main memory is updated. The most significant bit of this field is used as a flag to indicate whether the vertex normal is invalidated and needs to be recomputed; i.e., when the heightmap values are modified.

Vertex representation in the video memory stores the following attributes:

- Position of the vertex as a 3D floating point vector (12 bytes).
- Normal of the vertex as a 3D floating point vector (12 bytes).
- Color of the vertex compressed to a 4D unsigned byte vector (four bytes).
- Lower LOD transition position of the vertex as a 4D floating point vector (16 bytes). The w-component of this (x, y, z, w)-vector stores the transition distance.
- Lower LOD transition normal of the vertex as a 3D floating point vector (12 bytes).

3.2. *Vertex Index Calculation*

The presented approach does not require explicitly storing triangle vertex indices. Algorithm 1 computes triangle vertex indices on the fly with the help of a static lookup table. It should be noted that each vertex is shared by multiple triangles; not only by those belonging to the same patch but also by triangles of different patches (through shared vertex lists). It is now possible to render the terrain surface using a hardware-accelerated rasterization-based renderer by sending the vertices to the GPU and using triangles as the rendering primitive.

3.2.1. Surface Generation

Algorithm 2 finds out the virtual sub-voxels on which the surface extraction algorithm will work by handling voxel level transitions. Algorithm 2 calls the surface extraction algorithm (Algorithm 3) for each virtual sub-voxel.

Algorithm 3 generates surface patches for a single intersection volume. The generated surfaces are added to the list of Bézier surfaces stored in one of the voxels in the corresponding volume. We store surfaces generated in the voxel whose index fields have minimum values. Where they are stored does not matter as long as the choice is consistent and it ensures that the adjacent voxels store the surfaces for the adjacent volumes.

Algorithm 1: Generating triangle vertex indices

```
generateTriangleVertexIndices(triangleIndex,
                    numVerticesPerEdge, vertexIndices);
triangleIndex        : (input) Zero-based index of the triangle
numVerticesPerEdge: (input) Number of vertices per edge (N)
vertexIndices        : (output) Vertex indices of the triangle
begin
    const indices[8][3][2] = {
        { {0, 0}, {0, 1}, {1, 1} }, // face-1
        { {1, 1}, {0, 1}, {0, 2} }, // face-2
        { {0, 2}, {1, 2}, {1, 1} }, // face-3
        { {1, 1}, {1, 2}, {2, 2} }, // face-4
        { {2, 2}, {2, 1}, {1, 1} }, // face-5
        { {1, 1}, {2, 1}, {2, 0} }, // face-6
        { {2, 0}, {1, 0}, {1, 1} }, // face-7
        { {1, 1}, {1, 0}, {0, 0} }, // face-8
    };
    // % is the modulus operator; << is the left-shift operator;
    // >> is the right-shift operator; & is the bitwise-and operator.
    numFacesPerEdge = numVerticesPerEdge >> 1;
    i = (batchIndex % numFacesPerEdge) << 1;
    j = (batchIndex / numFacesPerEdge) << 1;
    k = triangleIndex & 0x7;
    for v ← 0 to 3 do
        vertexIndices ← (i + indices[k][v][0], j + indices[k][v][1]);
```

Algorithm 2: Generating surfaces for sub-voxel intersection volumes by handling voxel level transitions

```
generateSubVoxelSurfaces(root, patchCollection)
root              : (input) The root of the voxel representation
patchCollection: (output) The list of Bézier surface patches
begin
    if root has children then
        // construct patches for each child
        for i ← 0 to 8 do
            generateSubVoxelSurfaces(root.getChildVoxel(i), patchCollection);
        return;
    rootVoxelIndex = root.getIndex();
    levelDiff = maxLevel - rootVoxelIndex.level;
    //  << is the left-shift operator and  |  is the bitwise-or operator
    numSubVoxelsPerAxis = 1 << levelDiff;
    maxSubVoxelIndexPerAxis = (1 << (levelDiff)) - 1;
    for x ← 0 to numSubVoxelsPerAxis do
        for y ← 0 to numSubVoxelsPerAxis do
            for z ← 0 to numSubVoxelsPerAxis do
                // skip vertices that are not on surface of voxel
                if ((x  !=  0 and x  !=  maxSubVoxelIndexPerAxis) and
                    (y  !=  0 and y  !=  maxSubVoxelIndexPerAxis) and
                    (z  !=  0 and z  !=  maxSubVoxelIndexPerAxis)) then
                    continue;
                // compute voxel index of virtual child voxel
                subVirtualVoxelIndex.level = maxLevel;
                subVirtualVoxelIndex.xIndex = (rootVoxelIndex.x << levelDiff) | x;
                subVirtualVoxelIndex.yIndex = (rootVoxelIndex.y << levelDiff) | y;
                subVirtualVoxelIndex.zIndex = (rootVoxelIndex.z << levelDiff) | z;
                // construct patches for the virtual child voxels
                generateSurfacePatches(voxelIndex, patches);
                patchCollection.addPatches(patches);
```

Algorithm 3: Generation of patches for a single voxel

```
generateSurfacePatches(voxelIndex, patches)
voxelIndex: (input) The index of the voxel for which the surface patches will be generated
patches    : (output) The list of surface patches, stored per voxel
begin
    // construct the voxel intersection volume
    voxelIndices[8];
    for $i \leftarrow 0$ to 8 do
        //  >> is right-shift and % is modulus
        voxelIndices[i] = getNeighborVoxelIndex(i >> 2, (i >> 2) % 2, i % 2);
    // normalize voxel intersection volume
    normalizedVoxelGroups = normalize(voxelIndices);
    foreach voxelGroup in normalizedVoxelGroups do
        occupyFlag = 0;
        for $i \leftarrow 0$ to 8 do
            // |= is bitwise-or assignment and << is left-shift
            occupyFlag |= voxelGroup.isVoxelOccupied(i) << i;
        patches = computeControlPoints(occupyFlag);
        voxel = getVoxel(voxelIndex);
        foreach patch in patches do
            voxel.patches ← patch;
```

3.3. *Displacement of Terrain Vertices*

The displacement operation is performed on each vertex of all patches. However, due to shared vertices among patches, displacement may be applied multiple times on some vertices. To avoid this, we introduce the concept of vertex owners for externally shared vertices. The owner is the patch pointed to by the first node of the shared vertex list of that vertex. Displacement is applied on externally shared vertices only by the patch that owns the vertex.

The displacement operation can also occur multiple times on internally shared vertices, as multiple vertex indices can point to the same vertex if an edge of the patch is collapsed. In this case, displacement must be applied to only the first vertex index that points to the internally shared vertex. All other vertex indices that point to an internally shared vertex are called *inactive* indices, and displacement is not applied.

The displacement values of neighbor vertices should be similar to avoid jagged edges and sharp corners. To generate a smooth terrain surface, a Gaussian filter is applied while computing displacement values. Kernel size dramatically affects the resulting terrain: a large kernel size will over-smooth the surface by removing fine details and slow down the displacement operation, whereas a small kernel size will not be sufficiently effective in reducing jagged edges and corners. Our experiments have shown that a Gaussian kernel of size 3×3 usually yields sufficiently good results (see Figure 9).

Algorithm 4 applies displacement to the terrain surface vertices. Algorithm 5 finds the neighbor patches of the modified patch; Algorithm 6 updates the displaced positions of the vertices of the modified patch; and Algorithm 7 updates the vertex normals affected by the surface modification.

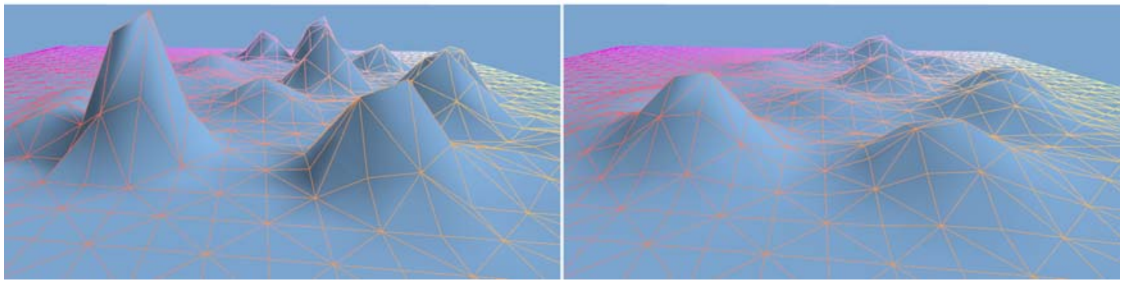

Figure 9. Left: the result of the displacement operation without Gaussian filtering. Right: the same displacement map applied with Gaussian filtering (kernel size is 3×3).

Algorithm 4: Displacement of terrain surface vertices

```
displaceVertices($octree$, $displacedPosition$)
octree           : ($input$) Octree that defines the terrain model
displacedPosition: ($output$) Displaced positions of each $vertex$
begin
    $uvScale = 1/(numVerticesPerEdge - 1)$;
    foreach $voxel$ in $octree$ do
        foreach $patch$ in $voxel$ do
            for $i \leftarrow 0$ to $numVerticesPerEdge$ do
                for $j \leftarrow 0$ to $numVerticesPerEdge$ do
                    $vertexIndex = \{i, j\}$;
                    if isVertexIndexActive($vertexIndex$) == $false$ then
                        continue;
                    if isBorderVertex($vertexIndex$) == $true$ then
                        if getVertexOwner($vertexIndex$) != $patch$ then
                            continue;
                    $u = i \times uvScale$;
                    $v = 1 - j \times uvScale$;
                    $displacementValue = patch.heightmap.getFilteredValue(u, v)$;
                    $vertex = patch.getVertexByIndex(vertexIndex)$;
                    $vertex.displacedPosition =$
                        $vertex.originalPosition + vertex.displacementNormal \times displacementValue$;
```

Algorithm 5: Computing neighbors of a patch

```
computeNeighbors($centerPatch$, $neighborPatches$)
centerPatch     : ($input$) The patch whose neighbors are queried
neighborPatches: ($output$) The set of neighbor patches of $centerPatch$, including the patch itself.
begin
    foreach $sharedVertexListNode$ in $centerPatch$ do
        $firstNode = sharedVertexListNode.firstNode$;
        while $firstNode$ != $NULL$ do
            $neighborPatches = neighborPatches \bigcup firstNode.surfacePatch$;
            $firstNode = firstNode.next$;
```

Algorithm 6: Updating the displaced positions of vertices

```
updateDisplacedPositions(modifiedPatch, displacedPosition)
modifiedPatch    : (input) The patch whose heightmap is modified
displacedPosition: (output) Displaced positions of vertices
begin
    $uvScale = 1/(numVerticesPerEdge - 1)$;
    foreach vertex in modifiedPatch do
        $vertexIndex = vertex.vertexIndex$;
        if isVertexIndexActive(vertexIndex) == false then
            continue;
        if isBorderVertex(vertexIndex) == true then
            if getVertexOwner(vertexIndex) != patch then
                continue;
        $u = vertexIndex.i \times uvScale$;
        $v = 1 - vertexIndex.j \times uvScale$;
        $displacementValue = patch.heightmap.getFilteredValue$(u, v);
        $vertex.displacedPosition = vertex.originalPosition +$
            $vertex.displacementNormal \times displacementValue$;
        // invalidate the vertex normal
        $vertex.recomputeNormal = 1$;
        // reset the vertex normal
        $vertex.actualNormal = \{0, 0, 0\}$;
```

Algorithm 7: Updating the affected vertex normals after editing a surface patch's heightmap

```
updateNormals(modifiedPatch, actualNormal);
modifiedPatch: (input) Surface patch whose heightmap is modified
actualNormal : (output) Updated normals of affected vertices
begin
    $neighborPatches = getNeighboringPatches(modifiedPatch)$;
    foreach patch in neighborPatches do
        foreach triangle in patch do
            foreach vertex in triangle do
                if $vertex.recomputeNormal == 1$ then
                    $triangle.recomputeNormal = 1$;
    foreach patch in neighborPatches do
        foreach triangle in patch do
            if $triangle.recomputeNormal == 0$ then
                continue;
            foreach vertex in triangle do
                $vertex.recomputeNormal = 1$;
    foreach patch in neighborPatches do
        foreach triangle in patch do
            $vertices = triangle.vertices$;
            if ((vertices[0].recomputeNormals == 0) and
                (vertices[1].recomputeNormals == 0) and
                (vertices[2].recomputeNormals == 0)) then
                continue;
            $\overrightarrow{v_1}$ = vertices[1] - vertices[0]
            $\overrightarrow{v_2}$ = vertices[2] - vertices[1]
            $\overrightarrow{N} = \overrightarrow{v_1} \times \overrightarrow{v_2}$
            for $i \leftarrow 0$ to 3 do
                if vertices[i].recomputeNormals == 1 then
                    vertices[i].actualNormal += $\overrightarrow{N}$
```

3.4. *Voxel Selection Ray Computation for Generating the Coarse Terrain Model*

Computing the voxel selection ray that is used in generating the coarse terrain model is performed by a number of transformations applied to the 2D cursor position. Algorithm 8 computes that ray, given the cursor's coordinates, and it is then used to determine the closest intersecting voxel at the desired level of the octree.

Algorithm 8: Computing the start and end positions of the ray in world coordinates defined by the cursor

```
computeRay(cursorPos, viewport, camera, rayStart, rayEnd)
cursorPos: (input) 2D cursor position on application viewport
viewport  : (input) Viewport attributes used for rendering
camera    : (input) Perspective camera attributes for rendering
rayStart  : (output) Starting ray position in world coordinates
rayEnd    : (output) Ending ray position in world coordinates
begin
    // compute normalized cursor coordinates
    $normalizedCursorX = (cursorPos.X \ / \ viewport.width) \times 2 - 1$;
    $normalizedCursorY = 1 - (cursorPos.Y \ / \ viewport.height) \times 2$;
    if $viewport.width > viewport.height$ then
        $normalizedCursorX$ *= $viewport.width \ / \ viewport.height$;
    else
        $normalizedCursorY$ *= $viewport.height \ / \ viewport.width$;
    // the field of view is in radians
    $fovFactor = tan(camera.fieldOfView \times 0.5)$;
    $normalizedCursorX = normalizedCursorX \times fovFactor$;
    $normalizedCursorY = normalizedCursorY \times fovFactor$;
    // compute the starting position of the ray
    $rayStart = \{$
            $normalizedCursorX \times camera.nearClipDistance$,
            $normalizedCursorY \times camera.nearClipDistance$,
            $-camera.nearClipDistance$, 1};
    // compute the ending position of the ray
    $rayEnd = \{$
            $normalizedCursorX \times camera.farClipDistance$,
            $normalizedCursorY \times camera.farClipDistance$,
            $-camera.farClipDistance$, 1};
    // transform ray definition into world space
    $inverseViewMatrix = inverse(camera.viewMatrix)$;
    $rayStart = inverseViewMatrix \times rayStart$;
    $rayEnd = inverseViewMatrix \times rayEnd$;
```

4. Discussion

We discuss how the proposed approach performs compared to heightmap- and voxel-based approaches.

Expressiveness is defined as the ability to create interesting terrain features using a terrain representation. A heightmap samples the terrain in 2D, hence it has limited expressive power. It is not possible to represent volumetric terrain features such as caves, overhangs, arches, and even vertical cliffs using heightmaps; voxel-based representations, on the other hand, sample the true 3D space surrounding the terrain. With an infinitesimal voxel size, we can represent all possible terrains, and any other 3D object for that matter. Increasing voxel resolution, however, has serious practical implications such as high memory usage.

The proposed hybrid approach is a combination of voxel and heightmap representations in terms of expressiveness. It converges to a voxel representation as the resolution of the coarse voxel model increases, and converges to a heightmap representation as it decreases. However, it is not possible to render high-resolution voxel models in real time anyway. To increase overall efficiency, the proposed approach provides high voxel resolution where volumetric features are dense and complex, and benefits from the simplicity of heightmap approaches where such features are redundant.

Simplicity of the algorithms is important. Heightmaps and their algorithms are simple. Using voxels to represent volumes is also simple. It is possible to write simple algorithms that work on voxel data, however, it is not plausible to use them for real-time applications. Voxel data is usually huge, which increases algorithm complexity: they must page data in and out of the memory, generate relevant parts of the surface from the voxel model at every frame, compress and decompress voxel data, etc. Our approach is a heightmap representation applied on a coarse voxel model; we can thus use existing algorithms for heightmap-based regular grid representations with only minimal changes.

Efficiency is necessary in the processing and memory usage demanded by real-time terrain rendering. Heightmaps are unbeatable in this respect. Their memory requirements are extremely low, even for very large terrains, because almost all vertex attributes are implicit, making it redundant to store them. Given any heightmap, a GPU can efficiently create the entire geometry, including attributes such as vertex normals. One of our design goals was to benefit from the simplicity and efficiency of heightmap representations, and thus the terrain surface in our approach is similar to a regular grid heightmap surface. Our approach is obviously not as efficient as heightmaps are, but it is much more expressive, and in terms of memory usage and demanded processing power, it is efficient enough for a real-time terrain rendering application.

Voxel-based terrain representations usually require high amounts of memory when they are stored uncompressed and at a high resolution. Memory efficiency can be increased at the cost of processing power by compressing and decompressing the voxel representation as needed, but this is difficult in a real-time rendering application. Voxel representations cannot be directly used for rendering, and unlike heightmaps, the polygonal surface of a voxel model cannot be easily generated. Doing this in real-time and at every frame, such as for LOD and culling algorithms, is even more difficult. As a result, voxel representations are not popular in real-time applications.

Visual quality depends on methods used for lighting, texturing, and shadowing. It is easy to generate the terrain surface of a heightmap without visual artifacts; LOD management is also easy because the terrain surface is greatly constrained and the connections between vertices are well defined and simple.

Generating a polygonal surface for a volumetric representation is not easy. Relatively simple methods, such as the original marching cubes, may generate visual artifacts. LOD management is problematic because most surface extraction methods do not support a sampling grid with different resolutions in different areas. Even methods that use an LOD approach (e.g., Lengyel (2010)) cannot support smooth LOD transitions and the visual quality is thus significantly degraded. Using patches, our surface extraction method can generate a smooth terrain surface for a coarse volumetric representation with no artifacts. Our LOD approach supports smooth LOD transitions by geometry morphing.

Content creation can be performed by procedural techniques, manual editing, or a combination of both. Quasi-random terrain data is usually generated by procedural techniques, which can be controlled by a set of parameters, and fine details are then added by artists. Creating heightmaps is relatively simple, and procedural techniques to do so are available. With a heightmap, one can create realistic terrains in a controllable way using noise functions and can manually edit them with an image editor or with a specialized 3D heightmap terrain editor. However, it is very difficult, if not impossible, to create interesting terrain features using heightmaps.

Procedural content creation techniques are available for volumetric terrains, but these cannot easily create volumetric features such as mountains, hills, caves, and arches. Manually editing volumetric models requires turning voxels on or off, which necessarily simplifies any volumetric features. The proposed representation can support procedural techniques for volumetric terrains to create the coarse model and heightmaps to create the surface details. Manual editing is easy for both representations, but can sometimes be unintuitive due to the displacement normals of the terrain surface. This problem can be overcome by using a terrain editor specifically designed for the proposed representation.

Physics computations for interactions with the terrain (such as collision detection with creatures and vehicles) are also usually necessary. For some applications, it is desired to modify the terrain in runtime according to these interactions. Voxel terrain models can be easily edited by turning voxels on and off. Modifying these models may require regeneration of the entire surface, however, because most surface extraction approaches do not support local updates. Simple physics computations are usually efficient because checking whether a voxel is empty or not requires a fast lookup. Performing more advanced physics computations depends on the structure used to store volumetric data.

Although heightmaps can be easily edited, updates are limited because only vertex height can be changed. Physics queries such as collision detection are simple because a given position in the space maps one to four samples on the heightmap, and because heightmaps allow limited terrain configurations.

The proposed representation allows deformations at the patch level in real time but the coarse model cannot be modified in real time. Patch deformations are efficiently handled and only cause local changes on the terrain surface. The coarse voxel model can answer some queries but it is not reliable because it does not take into account heightmap displacements applied to patches.

References

Gustavson, S., 2005. *Simplex Noise Demystified* [online], Available from: `http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf`, [Accessed at 19 March 2014].

Ikits, M. and Magallon, M., 2012. *GLEW: The OpenGL Extension Wrangler Library* [online], Available from: `http://glew.sourceforge.net/`, [Accessed at 19 March 2014].

Lengyel, E., 2010. Voxel-Based Terrain for Real-Time Virtual Simulations. Thesis (PhD). University of California at Davis.

Silicon Graphics, Inc., 2012a. *GLSL: OpenGL Shading Language* [online], Available from: `http://www.opengl.org/documentation/glsl/`, [Accessed at 19 March 2014].

Silicon Graphics, Inc., 2012b. *OpenGL: Open Graphics Library* [online], Available from: `http://www.opengl.org/`, [Accessed at 19 March 2014].

Turner, D., Wilhelm, R., and Lemberg, W., 2012. *The FreeType Project - A Free, High-Quality, and Portable Font Engine* [online], Available from: `http://freetype.sourceforge.net`, [Accessed at 19 March 2014].

Woods, D., Weber, N., and Dario, M., 2012. *DevIL - A Full Featured Cross-platform Image Library* [online], Available from: `http://openil.sourceforge.net/`, [Accessed at 19 March 2014].