1. Introduction

Dynamically deformable terrain has long been a desired feature of game developers. Ultrabook™ devices and Windows 8* tablets open new attractive opportunities for interactivity, where touch screens and other sensors can be used to influence the terrain in new ways.

This sample demonstrates a technique that enables users to dynamically deform terrain using a touch screen. The two main goals of the sample are to demonstrate how to leverage the complete Ultrabook platform to render fully dynamic textured terrain and how to exploit touch screens in Windows 8 to enrich the user experience. Figure 1 shows the sample screen shot.

![Figure 1: Screen Shot of the Deformable Terrain with Touch Screen sample](image)

This sample is based on the previously published SNB terrain sample. It exploits the same hierarchical quad tree data structure, triangulation algorithm, and compact triangulation encoding scheme. Please refer to the original sample for more information on these topics. The following features are new to this release:

- Touch screen is exploited for greater user interaction
- Asynchronous terrain update operations (height map displacement, re-triangulation, etc.)
- Layer-based texturing improves visual fidelity
These features are described in detail in the following sections.

2. Using the touch screen

Handling WM_TOUCH windows message

When the user touches the screen, a WM_TOUCH message is generated by the system. When this message is handled by the windows procedure, the low-order word of wParam contains the number of touch points associated with the message and lParam contains a touch input handle that can be used in a call to GetTouchInputInfo() to retrieve detailed information about the touch points. After the message has been processed, the handle must be freed by a call to CloseTouchInputHandle(). Please refer to MSDN for more detailed information. The following code snippet handles touch messages and adds the screen coordinates of the touch point to the list of pending points (see CTouchTerrainSample::CPUTHandleWindowsEvent()).

```cpp
    case WM_TOUCH:
    {
        UINT numInputs = LOWORD(wParam);
        static std::vector<TOUCHINPUT> pInputs;
        pInputs.resize(numInputs);
        if (GetTouchInputInfo((HTOUCHINPUT)lParam, numInputs, &pInputs[0], sizeof(TOUCHINPUT)))
        {
            for (int iTap = 0; iTap < (int)numInputs; ++iTap)
            {
                // Convert the touch point into window coords
                POINT pt = { pInputs[iTap].x/100, pInputs[iTap].y/100 };
                MapWindowPoints(NULL, hWnd, &pt, 1);
                m_NewTapPositions.push_back( std::make_pair(pt.x, pt.y) );
            }
            CloseTouchInputHandle( (HTOUCHINPUT)lParam );
        }
        break;
    }
```

Reconstructing world space positions of touched points

To reconstruct the world space position of the point being touched, we have to transform the point’s projection space coordinates by the world-view-projection inverse matrix. Projection space x and y coordinates can be derived from the screen position as shown below:

```
PosPS.x = -1.f + 2.f * ((float)xPos + 0.5f) / (float)windowWidth;
PosPS.y = 1.f - 2.f * ((float)yPos + 0.5f) / (float)windowHeight;
```

To get the z coordinate, we need to access the depth buffer. Directly reading data from the GPU-residing depth buffer would require CPU-GPU synchronization, which would cause performance degradation. To eliminate this, the depths of points being touched are first copied to temporary 1x1 staging textures, one texture for each point. Texture cache is used to create and manage these auxiliary textures. On subsequent frames we attempt to map these textures using the D3D11_MAP_FLAG_DO_NOT_WAIT flag to eliminate stalls. If the map operation succeeds, the CTouchTerrainSample::ModifyTerrain() method is called to modify the terrain at the selected locations. Note that the world-view-projection matrix of the frame on which the touch event happened, not necessarily the matrix from the current frame, must be used to reconstruct world position. If the map operation fails, the map attempt is repeated on the next frame. CTouchTerrainSample::ProcessPendingModifications() method
performs the steps described above. The process of reconstructing a touch point’s world coordinates is illustrated in Figure 2: Reconstructing touch point world coordinates.

In DirectX® 11, only the whole depth buffer can be copied. As a result, we cannot directly copy the required depth value to a 1x1 texture. To get around this limitation, we first copy the depth buffer to a resource having the same dimensions and format but without the D3D11_BIND_DEPTH_STENCIL flag set, and then copy the required texel from this resource to a 1x1 staging texture.

3. Asynchronous terrain updates

As soon as we find at which location the terrain is to be deformed, we have to perform a number of operations: displace the height map, re-triangulate the affected region, update the data base, re-compute the normal map, etc. These tasks are time-consuming, especially re-triangulation, and performing all the required operations in the main thread would cause noticeable stalls. Thus it is reasonable to rebuild the triangulation asynchronously in order to not affect rendering performance. However, using the old triangulation for the modified region could be insufficient to accurately represent the new surface geometry. Imagine a flat area represented by a small number of large triangles and a small hole created through modification, which cannot be accurately modeled by large triangles. To solve this problem, we temporarily render modified patches using the full resolution
triangulation until an updated triangulation is ready. Thus, when the terrain is modified, some tasks are done immediately to show the effect of deformation, while time-consuming operations are postponed to asynchronous threads:

- The main thread immediately updates the patch height map on the GPU by rendering to texture, assigns full-resolution triangulation to the patch, and starts the asynchronous thread.
- The asynchronous thread updates the height map on the CPU, rebuilds the triangulation and encodes it, and performs other time-consuming operations. As soon as the updated triangulation is ready, the main thread starts using it for rendering the patch.

Each node of the quad tree has a list of pending modifications. When the terrain is modified, the new modification is added to the list. The lists are then asynchronously processed by worker threads. The workflow of the terrain modification is depicted in Figure 3.

![Figure 3: Updating terrain triangulation](image)

Note that a modified patch could be located at any level of the quad tree, not necessarily at the finest resolution level. If the patch is located at a coarser level, all affected patches up to the finest resolution level are visited and updated.

To implement asynchronous execution, each node of the quad tree can be assigned one of the following tasks:
- **Increase LOD** task is executed when the model is about to be refined. It creates the patch’s children at level n+1.
- **Decrease LOD** is executed when the model is about to be coarsened. It processes four sibling patches at level n+1 and replaces them with one coarse parent patch at level n. If some of the children are marked as modified, the task updates the parent patch (creates and encodes new triangulation, calculates new minimum/maximum patch height, updates bounding box, etc.). If no children are marked as modified, the task does nothing.
- **Recompress** task is executed when terrain is modified. It recursively traverses all the affected quad tree nodes starting from the modified patch and updates triangulation for each node. When the procedure reaches the finest resolution level, it also updates the height map and material mask.

The following rules are also enforced to ensure correct execution:

- A new task **cannot** be assigned to a node until the previous task at this node is completed.
- A decrease LOD task **cannot** be created in the following cases:
  - There are pending modifications assigned to one or more child nodes which have not been processed yet.
  - There is an Increase LOD task executing for one or more child nodes.
  - There is a Recompress task executing for one or more child nodes.
- If new modifications are assigned to some child nodes while a decrease LOD task is executing, the task is discarded after it is completed and is executed again after the modifications are processed.
- An Increase LOD task is not assigned to a node until all modifications are processed by a Recompress task.

If there are new modifications while the Recompress task is running, they are added to the list and processed by a new Recompress task as soon as the previous task finishes.

**4. Layer-based texturing**

Terrain texturing in this sample is done using a well-known, tile-based approach where a number of materials such as sand, grass, rock, and sand are tiled across the surface and blended using weights given by a global mask texture. Material textures are usually tiled many times to provide high frequency details, while the mask texture has much lower resolution and is responsible for global terrain appearance. The material mask is managed in the same way as the height map. In addition to a height map and normal map, each terrain has a material mask covering the same terrain region. When terrain is modified, the material mask is updated along with the height map in the asynchronous thread.

In this sample each material is assigned diffuse and normal maps. A 4-channel material mask is used and the weight of a fifth (base) material is implicitly defined as a complement to 1. The normal map is used to alter world space normals. All blending is performed in the pixel shader, which provides high quality
visual results. To implement smooth transitions between different levels of detail, the pixel shader implements morphing, as shown in Listing 1.

```c
float3 RenderPatchPS(RenderPatchVS_Output In) : SV_Target
{
    float4 SurfaceColor = 0;

    // Load material weights
    float4 MtrlWeights = g_tex2DMtrlMap.Sample(samLinearClamp, In.NormalMapUV.xy);

    // Load material weights from parent patch to implement morph
    float4 ParentMtrlWeights =
        g_tex2DParentMtrlMap.Sample( samLinearClamp, In.NormalMapUV.zw);
    MtrlWeights = lerp(MtrlWeights, ParentMtrlWeights, In.fMorphCoeff);

    // Normalize weights and compute base material weight
    MtrlWeights /= max( dot(MtrlWeights, float4(1,1,1,1)) , 1 );
    float BaseMaterialWeight = saturate(1 - dot(MtrlWeights, float4(1,1,1,1)));

    // Get diffuse color of the base material
    float4 BaseMaterialDiffuse =
        g_tex2DTileTextures[0].Sample(samLinearWrap, In.TileTexUV.xy);

    // Get tangent space normal of the base material
    float3 BaseMaterialNormal =
        g_tex2DTileNormalMaps[0].Sample(samLinearWrap, In.TileTexUV.xy);

    // Load material colors and normals
    [unroll]
    for(int iTileTex = 1; iTileTex < NUM_TILE_TEXTURES; iTileTex++)
    {
        const float fThresholdWeight = 3.f/256.f;
        MaterialColors[iTileTex-1] =
            MtrlWeights[iTileTex-1] > fThresholdWeight ?
            g_tex2DTileTextures[iTileTex].Sample(samLinearWrap, In.TileTexUV.xy) : 0.f;
        MaterialNormals[iTileTex-1] =
            MtrlWeights[iTileTex-1] > fThresholdWeight ?
            g_tex2DTileNormalMaps[iTileTex].Sample(samLinearWrap, In.TileTexUV.xy) : 0.f;
    }

    // Blend materials and normals using the weights
    SurfaceColor = BaseMaterialDiffuse * BaseMaterialWeight +
        mul(MtrlWeights, MaterialColors);

    float3 SurfaceNormalTS = BaseMaterialNormal * BaseMaterialWeight +
        mul(MtrlWeights, MaterialNormals);
    SurfaceNormalTS = normalize(SurfaceNormalTS*2-1);

    // Load world-space normal
    float3 Normal;
    Normal.xy = g_tex2DNormalMap.Sample(samLinearClamp, In.NormalMapUV.xy).

    // Load parent patch world-space normal
    float2 ParentNormalXY =
        g_tex2DParentNormalMap.Sample( samLinearClamp, In.NormalMapUV.zw).

    // Blend between child and parent patch normals
    Normal.xy = lerp(Normal.xy, ParentNormalXY.xy, In.fMorphCoeff);
    Normal.z = sqrt( max(1 - dot(Normal.xy,Normal.xy), 0.0) );
    Normal = normalize( Normal );
```
// Transform tangent space normal
float3 Tangent = normalize(float3(1,0,In.HeightMapGradients.x));
float3 Bitangent = normalize(float3(0,1,In.HeightMapGradients.y));
Normal = normalize( mul(SurfaceNormalTS, float3x3(Tangent, Bitangent, Normal)) );

// Compute diffuse illumination
float DiffuseIllumination =
    max(0, dot(Normal, g_LightAttribs.m_f4DirectionOnSun.xyz));
float3 lightColor = g_LightAttribs.m_f4SunColorAndIntensityAtGround.rgb;
SurfaceColor.rgb *= (DiffuseIllumination*lightColor +
    g_LightAttribs.m_f4AmbientLight.rgb);

// Add fog
float FogFactor = exp( -In.fDistToCamera * g_TerrainAttribs.m_fFogScale );
SurfaceColor.rgb = SurfaceColor.rgb * FogFactor +
    (1-FogFactor)*g_TerrainAttribs.m_f4FogColor.rgb;
return SurfaceColor;

Listing 1: Full code listing of terrain rendering pixel shader

Note that to improve rendering performance, diffuse and normal maps are accessed only if the
corresponding weight is greater than some small threshold.

5. Sample architecture
The sample consists of a number of components as shown in Figure 4.
The elevation (CElevationDataSource), material mask (CMtrlMaskSource), and triangulation (CTriangDataSource) data sources provide access to the corresponding type of data. Elevation and material mask sources have a Modify() method that updates the required data region. The triangulation data source provides a CreateAdaptiveTriangulation() method that constructs adaptive triangulation for the specified quad tree patch.

The core component of the sample is the block-based adaptive model (CBlockBasedAdaptiveModel). It performs all the tasks required to maintain and update the patch quad tree. It accesses the data sources to retrieve data and creates/destroys patches in the quad tree. It also manages all the asynchronous tasks. The object also maintains the list of optimal patches that are rendered by CAAdaptiveModelDX11Render. This component is also responsible for rendering bounding boxes, the terrain minimap, and quad tree structure on the map. The object creates and manages all the required shaders, states, buffers, and all other resources.

The Terrain patch object (CTerrainPatch) contains all the resources associated with the patch, the height map, the material mask, and the adaptive triangulation. The raw texture management is done by the D3D texture cache. The terrain patch object also contains methods that modify the data on the GPU: the render techniques, which displace the height map and update material mask.
6. Controlling the sample
The following controls can be used to control the sample:

- **Wireframe** checkbox turns wireframe overlay on/off.
- **Update model** checkbox enables/disable the model update. When it is unmarked, the model is not adapting to the camera location and triangulation updates are not performed.
- **Show bound boxes** check box shows/hides bounding boxes of the quad tree patches.
- **Adaptive triangulation** checkbox enables/disables using adaptive triangulation for rendering a patch. When it is unmarked, the full-resolution triangulation is used to render each patch.
- **Screen space threshold** slider controls the accuracy of the constructed terrain model.
- **Rock, White sand, Yellow sand, Gray sand, and Snow** buttons select corresponding material.
- **Displacement scale** slider controls the magnitude and disposition of modification. The leftmost location corresponds to the maximum negative displacement. The rightmost location sets the maximum positive displacement. The middle location sets no displacement. Only the material mask is modified in this case.
- **Size** slider controls the area affected by the modification.
- **Const size in screen space** checkbox forces the modifications to have constant size in screen space (when possible).

Figure 5 shows some different materials applied to the terrain surface.
The right mouse button can also be used to modify terrain surface, and the middle mouse button rotates the light.

Rendering a fully textured terrain at 1600 x 900 resolution takes less than 11 ms on Intel® HD Graphics 4000 hardware.

7. Resources


4. SNB terrain sample

5. Touch for Windows Desktop sample


Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.
The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, and Ultrabook are trademarks of Intel Corporation in the US and/or other countries.

Copyright © 2012 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.