

Programming for Real-Time Tessellation on GPU

Natalya Tatarchuk

Joshua Barczak

Bill Bilodeau

AMD, Inc.



(a) An untessellated character

(b) Character rendered with GPU tessellation

(c) An untessellated character close-up

(d) Tessellated character close-up

Figure 1. An example of using tessellation to render extremely detailed characters in close-up (b and d). On the left, the same character is rendered without the use of tessellation using identical pixel shaders and textures as the images on the right. While using the same geometry memory footprint, we are able to add high levels of detail for the tessellated character on the left.

Real-time shading has significantly increased in quality and complexity during the last decade. We have seen great strides in our ability to express lighting and shadowing details via pixel shaders. Normal mapping is a *de-facto* standard for any recently shipping game, regardless of the genre. Advanced titles such as Crytek's Crysis™ are creating complex surface effects for accurate representation of small-scale surface features and high-quality lighting via techniques such as parallax occlusion mapping [Tatarchuk06]. Nevertheless, realistic shading itself is not enough to create convincing representations of rich worlds on the screen. The need for true geometric detail has not completely gone away, even though we can express surface subtlety with a variety of per-pixel effects. The typical tell-tale signs giving away current games' low-resolution geometric representations are the coarse silhouettes.

A simple solution springs to mind – why not just increase the polygon count of the objects in the game? There are many distinct costs associated with explicit geometric representation, such as polygonal surfaces encoded in typical meshes. Meshes are an inherently expensive representation. We need to store a fair amount of data per-vertex (positions, texture coordinates, and animation data such as skinning weights and indices, for example). This is directly related to the associated memory footprint for the mesh objects – and thus increasing the pressure on already-scarce memory (as every game developer dearly knows). Furthermore, large meshes display

poor utilization of GPU vertex caches, and put a strain on vertex fetch and general memory bandwidth. Also, consider the expense of artist time; animating and texturing extremely large meshes (>1 million vertices) is a tedious and time-intensive process.

It is quite clear that the brute force solution of increasing polygon count for pre-authored objects is not a reasonable solution.

An alternative solution is suggested by the latest advances in GPU architecture. Recent generations of commodity GPUs such as the Microsoft® Xbox® 360 and ATI Radeon™ HD 2000, 3000, and 4000 series have shown tremendous improvements in geometry processing. These include unified shader architecture (introduced with Xbox 360), more dedicated shader units, and hardware tessellation pipelines. The latest graphics APIs (such as Microsoft Direct3D® 10 [Blythe06]) provide a convenient programmable model for data-recirculation between different passes. These methods can also be accessible via extensions in Direct3D 9 or by using the XDK API. Furthermore, with the introduction of upcoming graphics APIs such as Microsoft Direct3D 11 [KLEIN08] and [GEE08], tessellation and displacement mapping will be universally supported across all hardware platforms designed for that generation and, thus, solidify tessellation as the first-class citizen in the real-time domain. The upcoming generation of games, including those authored for Xbox 360, will use tessellation for extreme visual impact, high-quality rendering, and stable performance. A thorough understanding of how this technology works is the key to quick and successful adoption.

Hardware tessellation provides several key benefits that are crucial for interactive systems, such as video games:

- **Compression:** Using tessellation allows us to reduce our memory footprint and bandwidth consumption. This is true both for on-disk storage and for system and video memory usage, thus reducing the overall game distribution size and improving loading time. The memory savings are especially relevant to console developers with scarce memory resources. When using tessellation, we are specifying the surface topology, parameterization and animation data for the coarse control mesh. This mesh is authored to have a low amount of detail, and just to capture the overall shape of the desired object (Figure 2, top). High-frequency details such as wrinkles, bumps and dents are only modeled for the super high-resolution mesh (Figure 2, bottom) and are captured by the displacement map for the in-game assets. We can then combine rendering of this control cage with GPU tessellation and displacement mapping to greatly increase the overall amount of detail for the rendered asset (as shown in Figure 3). Figure 1 shows an example of this for a character from the AMD demo “Froblins” [Froblins08]. Table 1 demonstrates memory savings for the Froblin character used in a practical scenario in a complex, game-like, interactive system. Note that animation data is only stored for the control cage. This means that tessellation allows us to increase animation quality by storing more animation data (morph targets or bones) for each control cage vertex.
- **Bandwidth** is improved because, instead of transferring all of the vertex data for a high-polygon mesh over the PCI-E bus, we only supply the coarse mesh to the GPU. At render time, the GPU will need to fetch only this mesh data, yielding higher utilization of vertex cache and fetch performance. The tessellator directly generates new data which is immediately consumed by the GPU, without additional storage in memory.
- **Scalability:** Because of its recursive nature, subdivision naturally accommodates LOD rendering and adaptive approximation with a variety of metrics. Hardware tessellation allows game developers to implement a simple, flexible LOD management scheme, with no significant runtime cost, and little impact on existing content pipelines.

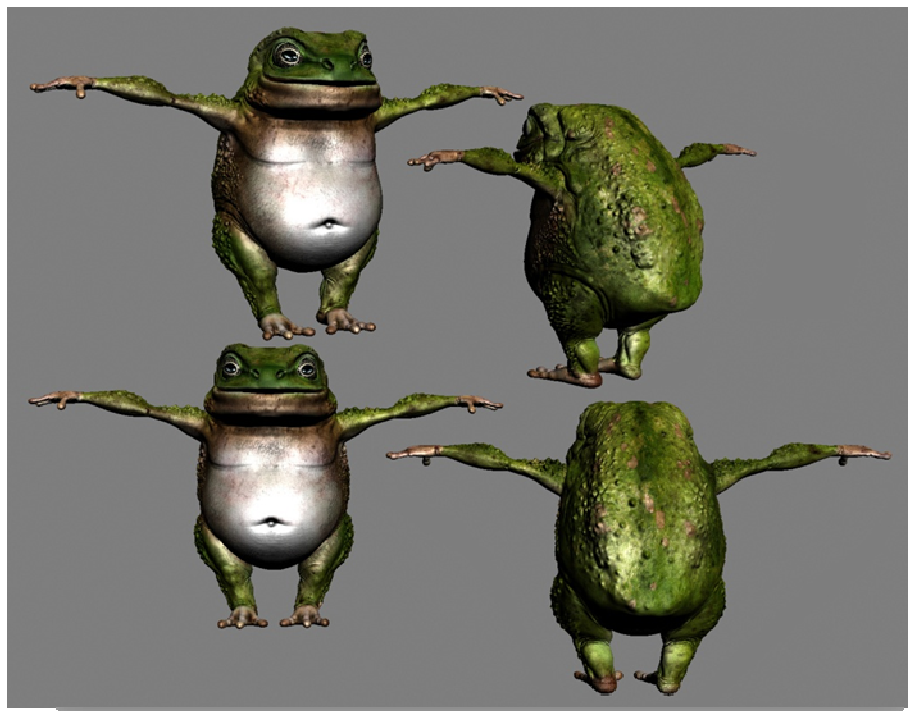


Figure 2. Comparison of low-resolution model (top, 5.1K triangles) and high-resolution model (bottom, 15M triangles) for the Froblin character.

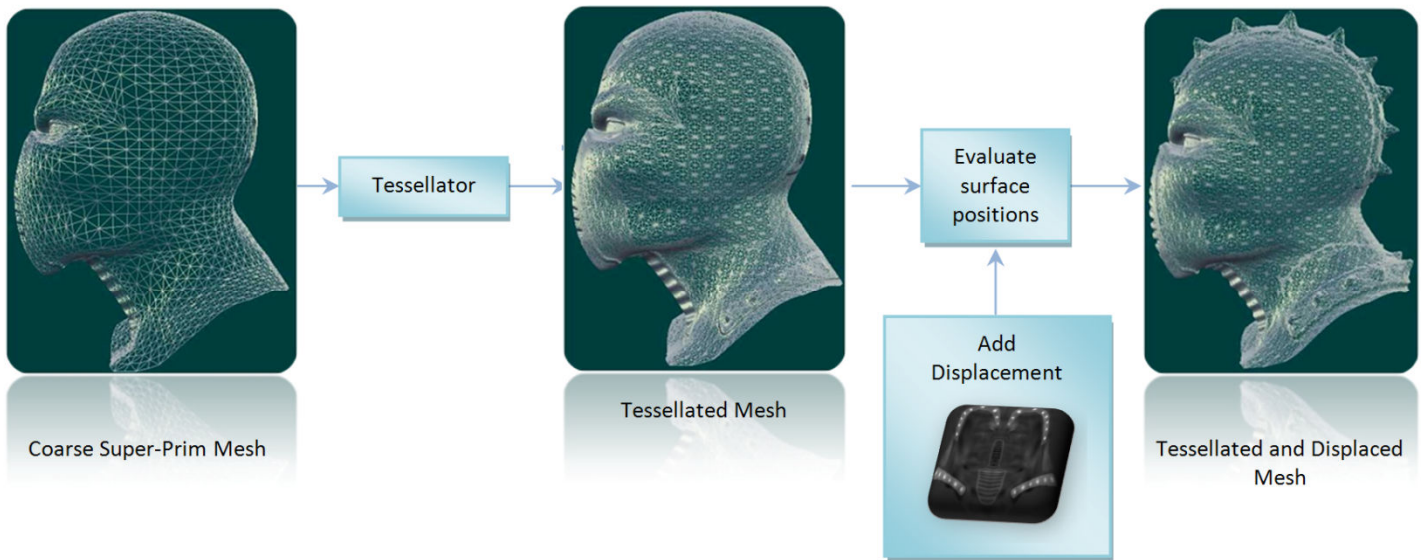


Figure 3. A simplified overview of the tessellation process. We start by rendering a coarse, low resolution mesh (also referred to as the “control cage” or “the superprimitive mesh”). The tessellation unit generates new vertices, thus amplifying the input mesh. The vertex shader is used to evaluate surface positions and add displacement, obtaining the final tessellated and displaced high-resolution mesh seen on the right.

	Polygons	Total Memory
Froblin control cage, Low-resolution model	5,160 faces	Vertex and index buffers: 100KB 2K × 2K 16 bit displacement map: 10MB
Zbrush® high-resolution Froblin model	15 M+ faces	Vertex buffer: ~270 MB Index Buffer: 180 MB

Table 1. Comparison of memory footprint for high and low resolution models for the Froblin character model.

GPU Tessellation Pipeline

Starting with the ATI Radeon HD 2000 series of graphics cards, PC developers can implement tessellation using dedicated hardware and a programmable pipeline. We designed an extension API for a GPU tessellation pipeline taking advantage of the hardware fixed-function tessellator unit available on recent consumer GPUs to access this functionality in Direct3D 9 and Direct3D 10/10.1 APIs. This tessellator is now available as a stage of the graphics pipeline across multiple APIs (Direct3D 9 through Direct3D 11 and Xbox 360) (Figures 4-6). We can compare how the graphics pipeline has changed with the addition of the tessellator by looking at the current graphics pipeline used by Direct3D applications.

The tessellation hardware is inserted before the vertex shader stage (Figure 4). The tessellator unit itself does not calculate the new vertex positions explicitly, nor does it store them in memory directly. Instead, this method calculates parametric or barycentric *domain* coordinates for new vertices that are passed to the vertex shader along with indices for the base primitive’s vertices. The vertex shader is responsible for calculating the position of the new tessellated vertices, thus *evaluating* the surface. This allows extremely fast generation of new surface points, as well as flexibility in the types of surfaces that we can evaluate. Despite the fact that the

tessellator unit is a fixed-function unit, using GPU tessellation we can in fact evaluate a number of different surface types ranging from first-order linear interpolation to higher-order surfaces such as Bezier or Catmull-Clark subdivision surfaces. Furthermore, the newly generated surface positions and their vertex attributes are not stored explicitly in video memory, which improves the overall performance.

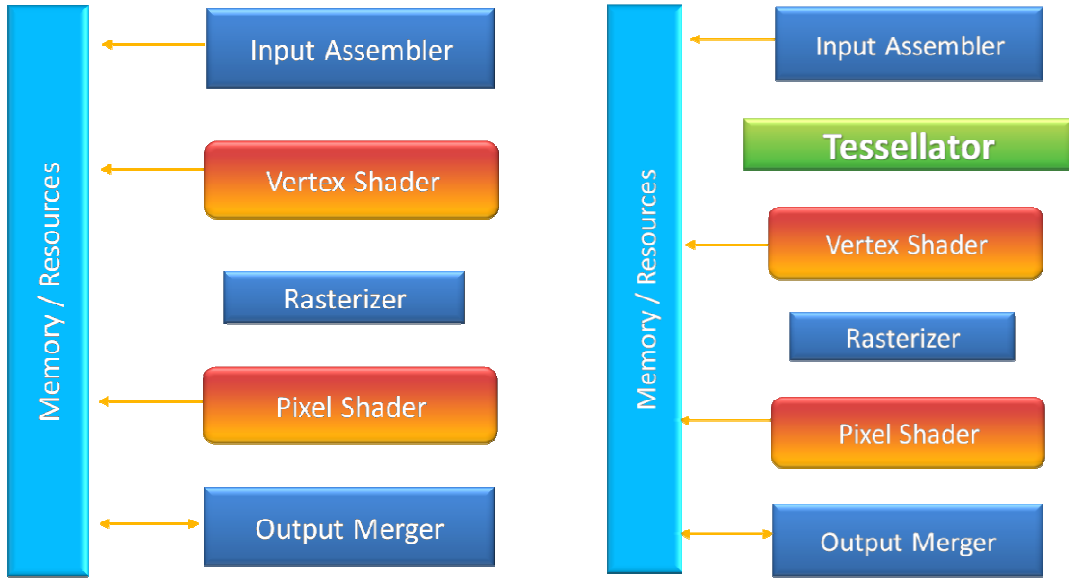


Figure 4. Comparison between the standard DirectX 9 graphics pipeline (left) and corresponding graphics pipeline with tessellation support (right, also representing Xbox 360 graphics pipeline). Note that sharp-corner boxes represent fixed-function units, and round corner boxes represent programmable stages.

Going through the process for the input polygonal mesh, as shown in Figure 3, we have the following: the hardware tessellator unit takes an input primitive (which we refer to as a *superprimitive*), and amplifies it (up to 411 triangles, or 15X for Xbox 360 or ATI Radeon™ HD 2000-4000 GPU generations, or 64X for Direct3D 11 generation of graphics cards). We refer to the input primitives as “superprimitives” or “superprims” because each superprim expands into many smaller triangle primitives when run through the tessellator. The API supports triangle and quad primitives. A vertex shader (which we refer to as an *evaluation shader*) is invoked for each tessellated vertex. This vertex shader receives the vertex indices of the superprimitive vertices and the parametric coordinates of the vertex being processed by this shader. In the case of triangles, these parametric coordinates are the barycentric coordinates, and for all other primitive types, the shader receives the parametric *uv* coordinates on the surface domain. The evaluation shader uses this information to calculate the position of the tessellated vertex, using any number of applicable techniques. This means that the vertex shader will typically need to convert from barycentric or parametric coordinates to world space coordinates before it does the usual world space-to-clip space transformation. We can use the following equations to convert from barycentric to world space coordinates for triangle superprims:

$$\begin{aligned}
 x &= x_1 v_x + x_2 v_y + x_3 v_z \\
 y &= y_1 v_x + y_2 v_y + y_3 v_z \\
 z &= z_1 v_x + z_2 v_y + z_3 v_z
 \end{aligned}$$

where (v_x, v_y, v_z) are the barycentric coordinates of the new vertex generated by the tessellator, (x_1, y_1, z_1) , (x_2, y_2, z_2) , and (x_3, y_3, z_3) are the triangle superprim vertices, and (x, y, z) is the resulting world space coordinate. For quads, use these equations to convert from parametric space to world:

$$x = x_l v_x + x_2 v_y + x_3 v_z + x_4 v_w$$

$$y = y_l v_x + y_2 v_y + y_3 v_z + y_4 v_w$$

$$z = z_l v_x + z_2 v_y + z_3 v_z + z_4 v_w$$

where (v_x, v_y, v_z, v_w) is the new vertex generated by the tessellator, (x_l, y_l, z_l) , (x_2, y_2, z_2) , (x_3, y_3, z_3) , and (x_4, y_4, z_4) are the quad superprim vertices, and (x, y, z) is the resulting world space coordinate.

The level of tessellation can be controlled either by a per-draw call tessellation factor (for continuous tessellation, as described below), or by providing per-edge tessellation factors in a vertex buffer for each triangle edge in the input mesh (for adaptive tessellation). For the latter, we can compute the edge factors using Direct3D 9 API by using the ATI R2VB extension [Persson06].

With Direct3D 10 API and beyond, we can combine tessellation with several additional features, such as geometry shaders, stream out, and instancing (Figure 5). This lets us compute per-edge tessellation factors with geometry shaders and stream out. We can also use these features for an animation and transformation pass for the low-resolution control cage. Additionally, should we wish to use the geometry shader post-tessellation (perhaps to compute per-face quantities, or to output to multiple render targets for cubemap rendering or reflections), the new pipeline supports this functionality.

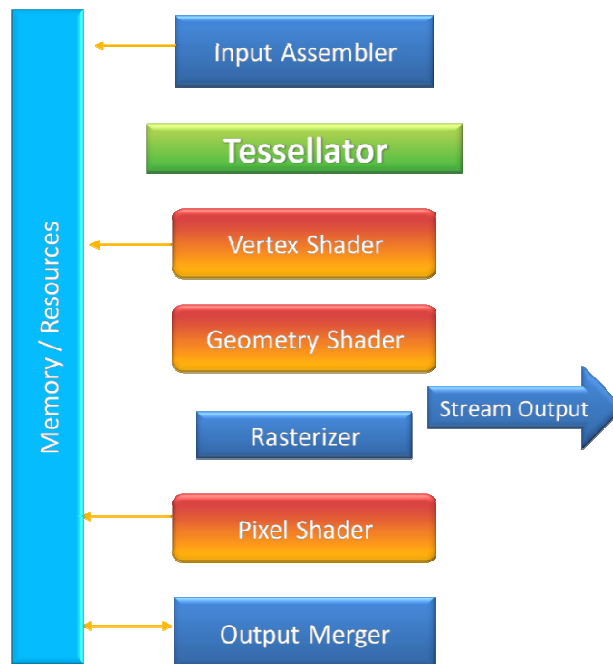


Figure 5. Direct3D 10 and 10.1 pipelines with tessellation support

The Direct3D 11 pipeline adds three new programmable stages to support tessellation plus a new compute shader (Figure 6). We easily map the methods described in this article to this pipeline, as our current tessellation API along with data re-circulation (using appropriate mechanisms in Direct3D 9 or 10) is a close subset of the full Direct3D 11 tessellation support. Therefore, game developers can use the cross-section of features to implement tessellation support across multiple APIs and operating systems.

With this programmable model, we start by rendering our mesh and executing the vertex shader. In the Direct3D 11 pipeline, the vertex shader is used to operate on the original control mesh vertices, performing control cage transformation and animation for the original mesh. Note that, at this point, the control cage mesh can be specified using any primitive format (be it patches or polygons).

In Direct3D 11, the *hull* shader is executed once per control primitive (the input primitive from the coarse control mesh). This can be used to implement per-patch operations. The hull shader can efficiently fetch all per-patch data and can be used to compute patch tessellation factors. Note that we can access the data for the entire primitive in the original mesh. We can also perform domain conversion in the hull shader. This is useful when we wish to convert some source mesh stored using one representation (for example, Catmull-Clark patches) to another representation, better suited for fast hardware evaluation (such as approximating those Catmull-Clark patches with Bezier patches – in this case, the hull shader can transform the control cage into the Bezier domain [LoopSchaefer08]). We can also do per-patch transformations, such as projecting patches into screen-space.

Next, the data is directed to the fixed-function tessellation unit. As in the Direct3D 9 and 10 pipelines, with tessellation support, this fixed-function unit generates *texture coordinates* and connectivity information for the domain primitive and amplifies the original data, up to 64X in Direct3D 11 and up to 15X in the DirectX 9 and 10/10.1 APIs. The amplification rate is important because it provides guidelines for the coarse control mesh polygon budget with respect to the desired features in the super high-resolution mesh (also referred to as the “limit surface”). If we want to represent high-frequency features in our high-resolution surface (such as small bumps, warts, wrinkles and bevels), we need to make sure that the coarse mesh has enough polygons to generate positions for these features. Therefore, the higher the amplification rate, the lower the polygon count for our control cage can be. Finally, the *domain* shader evaluates and outputs each surface point’s position, given patch ID and the generated *uv* coordinates. At this point, patch data is stored in on-chip storage.

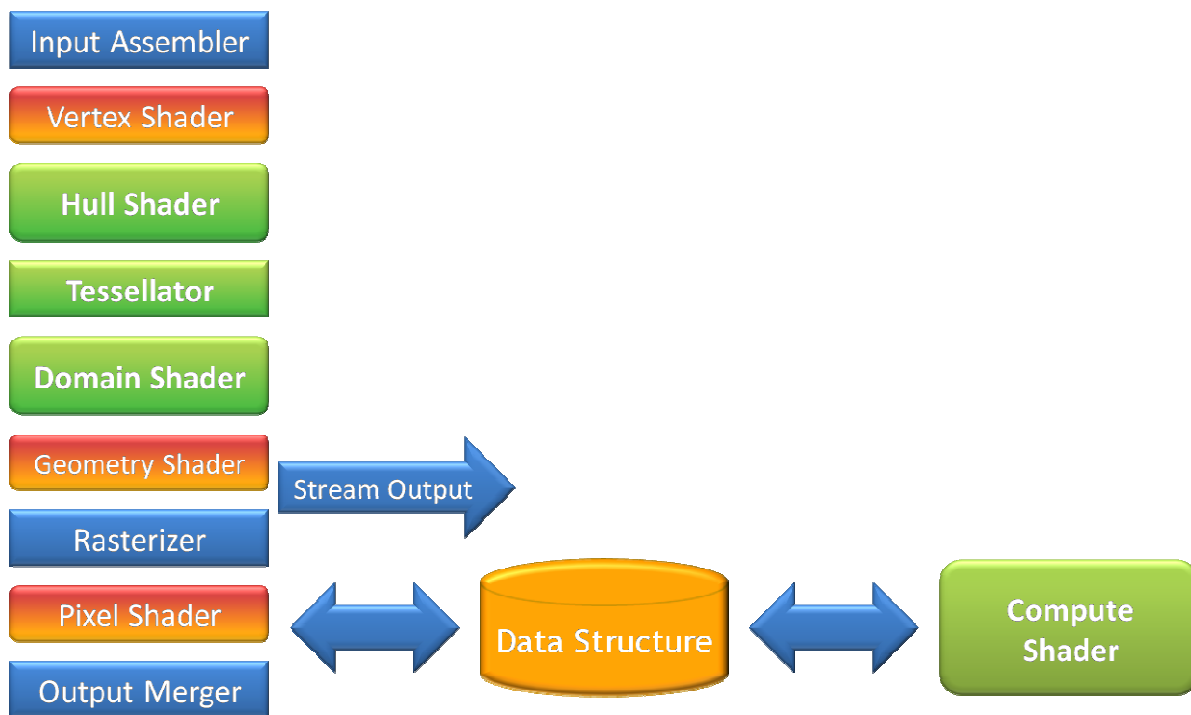


Figure 6. Direct3D 11 rendering pipeline.

Programming for GPU Tessellation



While Xbox 360 and Direct3D 11 on the PC explicitly support tessellation in their API models, we extended the Direct3D 9 through Direct3D 10.1 APIs to provide access to the tessellation functionality via a wrapper API, known as the ATI GPU Tessellation library. This library provides a lean and flexible interface to the tessellator hardware available on ATI Radeon GPUs. The library itself, as well as its programming guides, are available at [AMD09]. In this article, we’ll focus on the details of accessing the GPU tessellation functionality through the ATI

GPU tessellation library. We will describe the details of accessing the interface for Direct3D 9 API, but the interested reader can read more details on how to program Direct3D 10 API at [AMD09] at a later day.

The application starts by creating an instance of `IATITessellationD3D9` interface via the factory creation method, provided an existing D3D9 object and a D3D9 device:

```
IATITessellationD3D9* Create( IDirect3D9* pD3D, IDirect3DDevice9* pD3DDevice );  
  
IATITessellationD3D9* sg_pTessInterface = IATITessellationD3D9::Create( sg_pD3D, sg_pD3DDevice );
```

Code Listing 1. Creating a D3D9 tessellation interface instance using ATI GPU Tessellation library

The application will use this interface to access the tessellation extension. Therefore the application should create the instance of `IATITessellationD3D9` interface on start-up and maintain this instance for the duration of the lifespan of the associated D3D device. If the device is lost or reset, the application needs to delete the associated tessellation interface instance (by using the usual C++ constructs) and re-create it using the creation method just described.

Note that this creation method will query GPU tessellation support on the specified device, as well as perform the necessary interface initialization methods if GPU tessellation is supported. If this creation function returns a non-NULL valid interface object, the extension interface and GPU tessellation are supported on the given Direct3D9 device. A NULL return would signify that GPU tessellation is not supported on the provided device, and cannot be used.

There also additional methods for querying for tessellation support of specific modes and features using the `IATITessellationD3D9` interface:

```

//=====
/// Check tessellation support and query particular features. Once the query has been completed
/// simply returns the result for subsequent requests for a faster result.
/// \return True if tessellation is supported, false otherwise
//=====
bool IsTessellationSupported();

/// Queries for support for a specific tessellation mode. Returns true if the mode is supported,
/// false otherwise
//=====
bool CheckTessellationModeSupport( TSMode eMode );

//=====
/// Queries for support for a specific primitive type with tessellation for a given tessellation
/// mode and indexing mode for the draw call.
///
/// \param eMode
///         Tessellation mode
/// \param ePrimType
///         Primitive type we're querying support for
/// \param bIndexed
///         Specifies whether we're querying for an indexed (=true) or a non-indexed (=false) draw call.
///
/// Returns true if the primitive type is supported for the given tessellation mode for the specified
/// indexing method.
//=====
bool CheckPrimitiveTypeSupport( TSMode eMode, TSPrimitiveType ePrimType, bool bIndexed );

//=====
/// Queries whether unlimited vertex fetch feature combined with indices retrieval is supported
/// in the given tessellation mode.
///
/// \param eMode
///         Tessellation mode
///
/// Returns true if supported and false otherwise.
//=====
bool CheckUnlimitedFetchSupport( TSMode eMode );

```

Code Listing 2. *Methods for checking support for specific tessellation features*

Once a tessellation interface instance has been created, we can use it to set up the related rendering state and issue draw calls. There are two supported tessellation-rendering modes: continuous and adaptive. These modes define how new vertices will be created during tessellation for each domain primitive. We can set the rendering mode for a given draw call by calling `IATITessellationD3D9::SetMode(..)` method:

```

//=====
/// Enable tessellation and set active tessellation mode for subsequent draw-calls.
/// \param eMode
///         Tessellation mode parameter, specifies what type of tessellation mode to enable
///         or whether tessellation is disabled
/// \param dwTessFactorVBSourceIndex
///         Optional parameter. This parameter specifies the index of the stream source containing
///         tessellation factors during adaptive tessellation draw call. This parameter is ignored
///         when enabling continuous tessellation or disabling tessellation.
///         Default value is 1, where stream 0 will contain the regular vertex information.
///
/// \return true if tessellation is supported and the set call is successful, false otherwise
//=====
bool SetMode( TSMode eTessMode, DWORD dwTessFactorVBSourceIndex = 1 ) = 0;

```

Code Listing 3. *A method for setting tessellation mode during rendering*

with the following mode values: `TSMD_ENABLE_CONTINUOUS`, `TSMD_ENABLE_ADAPTIVE`, and `TSMD_DISABLE`. A typical sequence for rendering with tessellation will be:

- Enable tessellation by setting up a specific tessellation mode via `IATITessellationD3D9::SetMode(..)`

- Set up tessellation levels (min/max) as appropriate via `IATITessellationD3D9::SetMaxLevel(..)` and `IATITessellationD3D9::SetMinLevel(..)`
- Set up all rendering state (shaders, textures, parameters).
 - The associated vertex shader needs to be a correct evaluation shader (see following details)
- Issue draw calls through the tessellation interface by calling any of the methods below
- Disable tessellation by calling `IATITessellationD3D9::SetMode(TSMD_DISABLE)`

The tessellation library supports a number of draw calls, closely matching Direct3D API. Note that the SDK also includes the headers from the library with detailed documentation on the use of parameters for these methods:

```
HRESULT DrawIndexed( TSPrimitiveType ePrimType, uint32 nBaseVertexIndex, uint32 nMinVertex,
                    uint32 nNumVertices, uint32 nStartIndex, uint32 nNumPrimitives );

HRESULT DrawNonIndexed( TSPrimitiveType ePrimType,
                       uint32 nStartVertex,
                       uint32 nNumPrimitives );

HRESULT DrawMeshSubset( ID3DXMesh* pMesh, uint32 nSubset );
```

Code Listing 4. Methods for issuing draw calls for rendering tessellated characters

The tessellation library supports more primitive types than Direct3D 9 (such as quads). Therefore, to render objects using `DrawIndexed()` or `DrawNonIndexed()` methods, we need to specify the associated primitive type (for example, `TSPT_TRIANGLELIST`, `TSPT_TRIANGLESTRIP`, or `TSPT_QUAD`).

Continuous Tessellation Mode

Continuous tessellation is used to specify a single tessellation level per draw call. The tessellation level is a floating point value in the range of [1.0; 14.99] in Direct3D 9 and 10 APIs. The tessellation level is the amplification factor for the tessellated mesh.

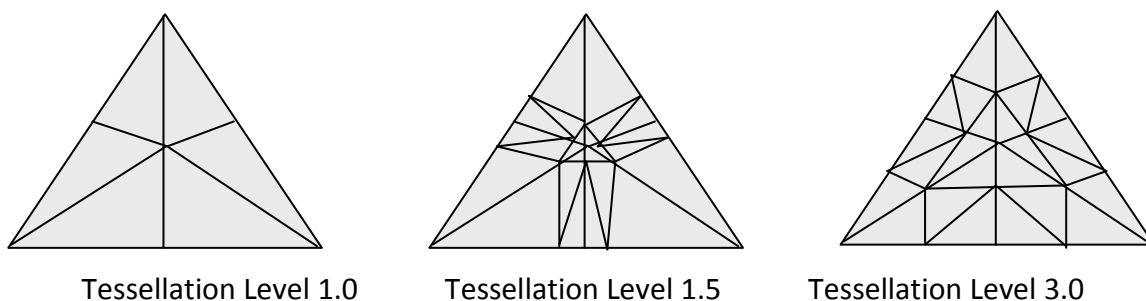


Figure 7. Continuous tessellation of a triangle superprimitive with increasing tessellation levels.

Continuous tessellation allows a smooth change to the level of detail depending on the object's position in the scene or other criteria. By using a floating point number for the tessellation level, we can gradually change the number of vertices and their positions to avoid popping artifacts. New vertices are only added immediately after each odd tessellation level. Notice how, in Figure 7, when the tessellation level is increased from level 1.0 to level 1.5, pairs of vertices are added near the center vertex of each edge, creating thin triangles close to existing edges. When we change from level 1.5 to level 3.0, we are not adding new vertices or triangles; we are just spreading the positions of the new vertices away from the center vertices along the edges. The effect of this fractional increase in the tessellation level is that the shape of the object changes gradually, instead of abruptly changing from a low level of detail to high level of detail, which would cause popping. With continuous tessellation, vertices and triangles are created in such a way as to ensure watertight tessellation. Notice that

when using continuous tessellation, the entire object is tessellated uniformly. While this can easily create an extraordinary amount of additional polygons, combining continuous tessellation with different object-based LOD schemes provides excellent tradeoff for fast, high-quality rendering. We can use a number of LOD heuristics to control the tessellation level based on object's proximity to camera, object type, etc. The SDK sample on the accompanying DVD named "CharacterRendering" in the directory for this chapter provides an example of computing the tessellation level based on object's distance to camera.

To set up continuous tessellation for a given draw call, we need to enable this mode (by calling `IATITessellationD3D9::SetMode(..)`). We also need to specify the *maximum* tessellation level by calling `IATITessellationD3D9::SetMaxLevel(..)` method. This is the tessellation level that will be used for rendering the object.

```
// Enable tessellation:
sg_pTessInterface->SetMode( TSMD_ENABLE_CONTINUOUS );

// Set tessellation level:
sg_pTessInterface->SetMaxLevel( 7.8f );
```

Code Listing 5. *Example draw call for rendering an object with continuous tessellation*

We then need to make sure that the shaders used for the draw calls are the appropriate evaluation shaders (see details in Listing 6). For example, we may wish to render our object with displacement mapping and with or without wireframe display. This code snippet from the included "CharacterRendering" SDK sample shows just that:

```
// Select appropriate technique to render our tessellated objects:
if ( sg_bDisplaced )
    if ( sg_bWireframe )
        sg_pEffect->SetTechnique( "RenderTessellatedDisplacedSceneWireframe" );
    else
        sg_pEffect->SetTechnique( "RenderTessellatedDisplacedScene" );
else
    if ( sg_bWireframe )
        sg_pEffect->SetTechnique( "RenderTessellatedSceneWireframe" );
    else
        sg_pEffect->SetTechnique( "RenderTessellatedScene" );

UINT iPass;
UINT cPasses;

V( sg_pEffect->Begin( &cPasses, 0 ) );
for ( iPass = 0; iPass < cPasses; iPass++ )
{
    V( sg_pEffect->BeginPass( iPass ) );
    if ( sg_pTessInterface )
    {
        V( sg_pTessInterface->DrawMeshSubset( sg_pMesh, 0 ) );
    }
    V( sg_pEffect->EndPass() );
}
V( sg_pEffect->End() );

// Compute the number of faces drawn per-frame for the tessellated object post-tessellation:
sg_nNumTessellatedPrims =
    sg_pTessInterface->ComputeNumTessellatedPrimitives( TSPT_TRIANGLELIST, sg_pMesh->GetNumFaces() );

// Disable tessellation:
sg_pTessInterface->SetMode( TSMD_DISABLE );
```

Code Listing 6. *Example render path for rendering a character with continuous tessellation using ATI GPU tessellation library and D3D9 FX framework.*

Rendering Characters with Tessellation

We can render characters using a number of different surface formulations, ranging from linear interpolative subdivision through higher-order surfaces. There exist a number of different subdivision methods, such as cubic Bezier patches; approximation via N-Patches [VPBM01]; B-splines and NURBs [PiegITiller96], often rendered with quads; Loop midpoint subdivision scheme used with triangles [Loop87]; and, Catmull-Clark subdivision surfaces [CatmullClark78, Stam98]. These schemes differ in the type of refinement rule (face or vertex split), the type of supported mesh (triangular or quadrilateral or mixed), whether the scheme is approximating or interpolating, and, finally, the smoothness of the limit surfaces (C^1 , C^2 , and so forth). A comprehensive overview of the various approaches can be found in [ZorinSchröder00]

In this chapter we describe how to render characters using *interpolating* subdivision. The key idea with this method is that the original mesh vertices remain undisturbed while the new points are inserted. Therefore, all vertices in the coarser LODs are maintained, and are also vertices of the refined meshes. With approximating subdivision, the coarse mesh approximates the final, subdivided surface. As we subdivide the mesh, we approach the final surface with each subdivision level (which is why it's often referred as the *limit* surface).

There are several advantages to the interpolating scheme. First, the original control points defining the low-resolution mesh are also the points on the limit (high-resolution) surface. This lets us control the limit surface in an intuitive manner. With approximating schemes (such as Catmull-Clark), the quality of the resulting surface is typically higher, and the approximating schemes converge to the limit surface faster than the interpolating schemes. However, the evaluation cost for an approximating scheme, is significantly higher (more than 100X) than that of a planar interpolating subdivision.

Traditionally, the vast majority of game titles support a wide range of graphics hardware platforms. This typically implies a range of API support, often ranging from fairly low specification versions (such as DirectX 8) all the way to DirectX 10.1. At the same time, game production budgets have exploded in recent years, primarily due to the cost of authoring art assets. Therefore, art asset re-use and consistency is very important for game developers, especially those targeting multiple platforms.

With interpolating subdivision, the art assets for the coarse control cage can be used to render the object on platforms that do not have support for GPU tessellation. Thus, the artists would only need to author one set of meshes, animation data and textures. These assets would have to be authored with tessellation and displacement mapping in mind, but they would successfully render on lower-end platforms. With the updated tool support for displacement map generation and tessellation preview (as in AMD GPUMeshMapper), artists have convenient methods for authoring for interpolating subdivision rendering. However, unless a game is designed to render entirely with subdivision surfaces, to include both subdivision surfaces and triangular meshes, the artists would need to generate two sets of art asserts. Subdivision surfaces are well supported in the digital content creation (DCC) tools such as Autodesk® Maya® or 3DStudioMax®. To reduce some of the burden of authoring dual art assets (animated triangular meshes for traditional rendering and animated control cages for subdivision surface rendering), artists can maintain some level of association from the original triangular meshes used to generate subdivision surface control cage throughout the creation process for game assets using these tools. At the same time, one would need to be conscious to maintain this association, especially when, for example, generating extreme displacement details by using an additional tool, such as Pixologic® Zbrush® or Autodesk Mudbox®. We found that, at this time, the association between the low-resolution model and super high-resolution model can easily be broken with the existing art authoring pipelines. However, a game wishing to support GPU subdivision surface rendering as well as traditional rendering would need to ship with both sets of art assets; therefore, this association needs to be factored into the art creation process early on.

Given the simplicity of implementation, fast evaluation and the convenience of authoring art resources, we find that interpolative planar subdivision combined with displacement mapping is an excellent approach for increasing surface quality for rendering highly detailed characters.

In our case, we render our characters using continuous tessellation. We specify the tessellation level per draw-call for a given LOD. Therefore, we can use tessellation to control how finely we are going to subdivide a character's mesh. We can use the information about character location on the screen or other factors to control the desired amount of detail. Furthermore, we use the same art assets for rendering the tessellated character as for the regular, conventional rendering used in current games.

Combining tessellation with instancing allows us to render diverse crowds of characters with minimal memory footprint and bandwidth utilization. In the "Froblins" demo, we render our characters by storing only a low-resolution model (5.2K triangles), and applying a displacement map in the evaluation shader. This lets us render a detail-rich, 1.6 million-triangle character using very little memory. Code Listing 9 provides an example of the vertex shader similar to the one used for evaluating the resulting surface positions. In our implementation, we combined Direct3D 10 instancing with GPU tessellation to render large crowds of highly detailed characters. The interested reader can find the exact shader used to render our Froblin characters in [SBOT08].

Designing Vertex Evaluation Shader for GPU Tessellation

To compute the surface position in the evaluation shader, the subdivision algorithms require access to the vertex data of the entire superprimitive (its control points). This is needed so we can combine the original primitive's vertex data (positions, etc.) using the selected evaluation method (via a control points stencil or by using barycentric interpolation, for example). In Direct3D 11, there are explicit API constructs in the hull and domain shaders to get access to the vertex data for the entire superprimitive (be it control points for a patch, or the vertices for a triangle). On the Xbox 360, we can use explicit vertex fetches to retrieve the data as needed.

In Direct3D 9 through 10.1, we need to author the vertex shader using a specific input structure format. This allows the tessellation interface to provide the superprimitive data correctly for the evaluation shader at run-time. There are several ways to access the superprimitive data in the evaluation vertex shader. We can start by using the vertex shader input data (either by declaring vertex shader input semantics in Direct3D 9 or by defining a vertex shader input structure in Direct3D 10/10.1). A regular vertex shader is set up to receive data for the single vertex that is being computed by this vertex shader. However, if, for example, we will render the tessellated mesh consisting of triangles, our vertex evaluation shader will need to access vertex data for all three vertices of the original mesh triangles.

The tessellation interface will provide the generated surface position coordinates as well as the superprimitive data to the evaluation shader whenever the tessellation is enabled by setting up either the continuous or the adaptive tessellation mode and an associated tessellation library draw call is used. However, unlike the typical vertex shader input declaration, we do not explicitly supply the data for superprimitives in the corresponding vertex streams' channels (which would require allocating additional storage in the vertex buffer). Instead, we render the tessellated mesh with the regular vertex buffer set up as we would use for traditional rendering (for example, with a vertex and an index buffer set up for indexed rendering). Then, during the tessellated draw call, the additional superprimitive vertex data is automatically inserted into the vertex shader input slots as declared in the evaluation vertex shader.

This is accomplished via explicit input mappings for specifying which superprimitive vertex data needs to be placed into which vertex shader input slots (or DECLs, using Direct3D 9 terminology).

Let's look at an example of modifying an existing Direct3D 9 vertex shader for traditional rendering to be used with tessellation.

```
float4x4 mWVP;
float4x4 mMW;

sampler2D sDisplacement;

uniform float fDisplacementScale;
uniform float fDisplacementBias;

struct VsInput
{
    float4 vPosition : POSITION0;
    float2 vTexCoord : TEXCOORD0;
    float3 vNormal : NORMAL0;
};

struct VsOutput
{
    float4 vPosCS : POSITION;
    float2 vTexCoord : TEXCOORD0;
    float3 vPositionWS : TEXCOORD1;
    float3 vNormal : TEXCOORD2;
};

VsOutput VS( VsInput i )
{
    VsOutput o;

    o.vTexCoord = i.vTexCoord;
    o.vNormal = i.vNormal;

    // Sample displacement map:
    float fDisplacement = tex2Dlod( sDisplacement, float4( i.vTexCoord, 0, 0 ) ).r;
    fDisplacement = (fDisplacement * fDisplacementScale) + fDisplacementBias;

    o.vPositionWS = i.vPosition + (fDisplacement * i.vNormal);

    // Transform position to clip-space
    o.vPosCS = mul( mWVP, float4( o.vPositionWS, 1.0 ) );
    return o;
}
```

Code Listing 7. *A simple Direct3D 9 vertex shader for rendering with displacement mapping.*

```

float4x4 mWVP;
float4x4 mMW;

sampler2D sDisplacement;

uniform float fDisplacementScale;
uniform float fDisplacementBias;

struct VsInput
{
    float3 vBarycentric: BLENDWEIGHT0;

    // Superprim vertex 0:
    float4 vPositionVert0 : POSITION0;
    float2 vTexCoordVert0 : TEXCOORD0;
    float3 vNormalVert0   : NORMAL0;

    // Superprim vertex 1:
    float4 vPositionVert1 : POSITION4;
    float2 vTexCoordVert1 : TEXCOORD4;
    float3 vNormalVert1   : NORMAL4;

    // Superprim vertex 2:
    float4 vPositionVert2 : POSITION8;
    float2 vTexCoordVert2 : TEXCOORD8;
    float3 vNormalVert2   : NORMAL8;
};

struct VsOutput
{
    float4 vPosCS      : POSITION;
    float2 vTexCoord    : TEXCOORD0;
    float3 vPositionWS : TEXCOORD1;
    float3 vNormal      : TEXCOORD2;
};

VsOutput VS( VsInput i )
{
    VsOutput o;

    // Compute new position based on the barycentric coordinates:
    float3 vPosTessOS = i.vPositionVert0.xyz * i.vBarycentric.x +
                       i.vPositionVert1.xyz * i.vBarycentric.y +
                       i.vPositionVert2.xyz * i.vBarycentric.z;

    // Compute new texture coordinates based on the barycentric coordinates:
    o.vTexCoord = i.vTexCoordVert0.xy * i.vBarycentric.x +
                 i.vTexCoordVert1.xy * i.vBarycentric.y +
                 i.vTexCoordVert2.xy * i.vBarycentric.z;

    // Compute new normal based on the barycentric coordinates:
    o.vNormal = i.vNormalVert0.xyz * i.vBarycentric.x +
               i.vNormalVert1.xyz * i.vBarycentric.y +
               i.vNormalVert2.xyz * i.vBarycentric.z;

    // Sample displacement map:
    float fDisplacement = tex2Dlod( sDisplacement, float4( o.vTexCoord, 0, 0 ) ).r;
    fDisplacement = (fDisplacement * fDisplacementScale) + fDisplacementBias;

    o.vPositionWS = vPosTessOS + ( fDisplacement * o.vNormal);

    // Transform position to clip-space
    o.vPosCS = mul( mWVP, float4( o.vPositionWS, 1.0 ) );
    return o;
}

```

Code Listing 8. Example of a simple Direct3D 9 evaluation shader for rendering tessellated characters with displacement mapping.

Notice the difference in the input structure declaration in the two listings (Code Listing 7 and Code Listing 8). The evaluation shader in Listing 8 has seven additional channels (which are not provided by the vertex buffer data explicitly). The programmer just needs to specify the expected data for each superprimitive vertex and the tessellation interface inserts this data during the draw call directly. So, to convert a regular vertex shader into a

GPU tessellation evaluation shader, we need to provide additional declarations for all of the input channels we wish to use for the input mesh’s vertices. When designing an evaluation shader, we need to specify enough additional channels for the primitive type we will be using for tessellated draw call. In the example in Code Listing 8, since we are rendering the mesh as a triangle list, we specify three sets of vertex input data. If we wanted to render with quads, we would specify four sets. Note that we can only use usage types that have several usage indices for the per-vertex superprimitive data. For example, position usage can be indexed 0..15 in Direct3D 9 (POSITION0..POSITION15), as with texture coordinates and so on. We can map each stream channel in the input declaration to the superprimitive vertex declaration as follows:

$$input_usage_index + superprimitive_vertex_index * 4,$$

regardless of the input primitive type.

The default mechanism uses the vertex shader DECL mapping to pass the superprim vertex data to the vertex shader using the USAGE index values 0, 4, 8, and 12. In other words, we can map the data for vertex *v*₀ to data channels with usage 0, the data for vertex *v*₁ to data channels with usage 4, and so on (as shown in the code listings above).

The parametric surface coordinates will be provided to the evaluation shader in the BLENDWEIGHT0 channel, which should be mapped to floating point values as you saw in the example above. If we want to access the indices of the super primitive vertices (which can be useful for accessing additional vertex data, or for tessellation edge factors computation), we can access them in the evaluation shader via the BLENDINDICES0 channel (note that the data will be typed as uint).

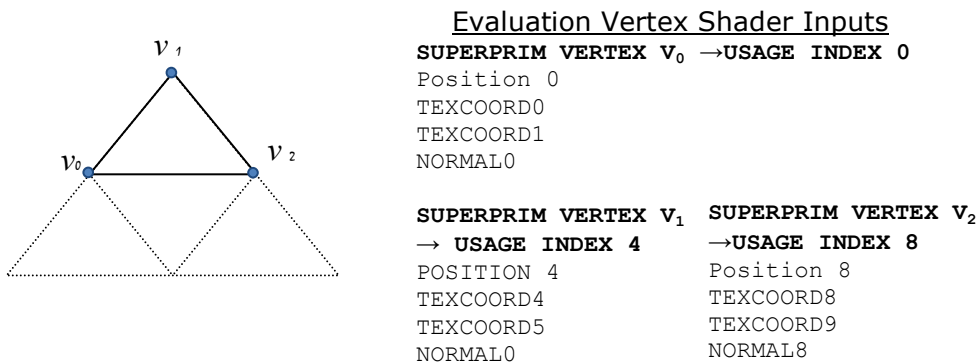


Figure 8. Rules for mapping from superprimitive vertex data to vertex shader inputs.

Accessing Per-Vertex Data Beyond Input Structure Declaration

One limitation of the Direct3D 9 and 10 APIs is that we can only access 16 float4 vertex inputs total in the evaluation vertex shader. Since one of these is dedicated to the parametric coordinates, we only have 15 float4 components to use for loading superprimitive vertex data in these APIs. This means that, as application developers, we must be mindful about data packing in order to bring in sufficient data. If we are rendering with tessellated triangles, this means that we can bring in at most 5 float4 components per superprimitive vertex. In the case of quads, we can only bring in 3 float4 components. This can be insufficient for some applications. This limitation is significantly relaxed for Direct3D 10.1 where we can fetch up to 32 vertex inputs directly.

We can work around these limitations by fetching the superprimitive data directly in the evaluation shader. In Direct3D 9, we can store the vertex buffer in an R2VB resource or a vertex texture and then fetch the data directly in the shader. In Direct3D 10, we can simply re-use the existing vertex buffer, mapping it as a shader resource for loading data in the shader. However, to load the per-vertex superprimitive data in the evaluation shader manually, we need to know the indices of the superprimitive vertices. The tessellation library provides this data to the evaluation shader whenever index retrieval mode is enabled via the following method of `IATITessellationD3D9`:

```
bool ToggleIndicesRetrieval( bool bEnable )
```

Note that this needs to be toggled per-draw call or just enabled for the duration of all rendering. Whenever index retrieval is enabled, the indices of the superprimitive vertices will be passed to the evaluation vertex shader into the `BLENDINDICES` semantic register. This allows the vertex shader to retrieve additional vertex elements by explicitly fetching them from the associated textures. The texture coordinates for these fetches can be computed from the super primitive vertex indices, and the textures themselves can easily be generated during asset preprocessing. Using this approach, it is possible to retrieve an arbitrary amount of data for each superprimitive vertex (though one must obviously be mindful of the performance implications of using too many fetches). We have included a Direct3D 9 tessellation SDK tutorial sample created by Nicolas Thibieroz that demonstrates the use of this mode.

Tessellation API in Direct3D 10

There are a number of improvements when using tessellation with DirectX3D 10 and 10.1 APIs due to increased API flexibility. While we will leave the details of ATI Tessellation library in Direct3D 10 to a later time, here are some important details:

- We can combine tessellation draw calls with instancing in Direct3D 10. This means that we can specify the tessellation level per instanced draw call and render an army of characters with a single mesh. We can combine this with texture arrays and other Direct3D 10 methods for creating a varied, interesting array of characters. We can also combine this method with the use of geometry shaders to perform LOD selection dynamically. For more details on this technique used in a practical scenario, see [SBCT08]
- In Direct3D 10, we can simply specify that the mesh's vertex buffer can be used as a texture for sampling data, and bind it as both for the tessellation draw-call; therefore, we don't need to explicitly copy the vertex buffer contents into a separate texture.
- We can use stream out post-vertex shader to compute per-edge tessellation factors and to perform control cage animation and transformation. We can also use geometry shader post-tessellation to either stream out the data for later use, to compute per-face quantities, or to perform further amplification (fur fins, render to cubemaps for reflections, etc.).
- In Direct3D 10.1, we also have an increased number of vertex inputs (32). This eliminates some of the pressure to use explicit vertex fetches.
- We have access to system-generated vertex IDs in these API versions' vertex shaders. This eliminates the need to generate this data channel during pre-processing and manually specify it to the shader.

Lighting with Tessellation and Displacement Mapping

Traditionally, animated characters are rendered and lit using tangent-space normal maps (TSNM). However, when we are rendering our characters with displacement mapping, to benefit from the additional details provided by the displacement maps, it is necessary to compute the lighting using the normal at the displaced

surface position (Figure 9). This necessitates that the normal stored in the TSNM is computed using the same art assets and tangent space computation as were used to compute the displacement map, ideally at the same time, to ensure that they match identically. This consistency is crucial for accurate lighting. In fact, the ideal authoring pipeline would generate the two maps at the same time, using a single tool. By using the publicly available AMD GPUMeshMapper [AMDGMM08] tool, which provides source code for tangent-space generation, we can ensure these requirements are met, and we can shade our characters using any lighting techniques that utilize TSNM. This is very important, as it allows us to use normal maps for animated character rendering (since world-space normal maps would be problematic for that purpose).

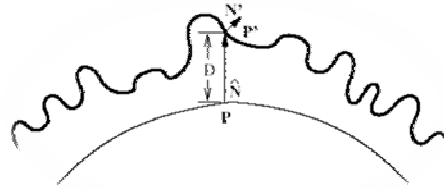


Figure 9. Displacement of the vertex modifies the normal used for rendering. $\mathbf{P}$ is the original point displaced in the direction of geometric normal $\hat{\mathbf{N}}$ displacement amount D . The resulting point $\mathbf{P}'$ needs to be shaded using normal $\mathbf{N}'$.

Rendering Animated, Tessellated Characters

When we render animated characters with subdivision, we need to perform animation calculations on the control mesh (the superprimitives), and then interpolate between the animated superprimitive vertices. A brute force approach of transforming and animating the superprimitive vertices in the evaluation shader wastes performance and bandwidth due to redundant computations – all newly generated tessellated vertices would perform the same computations as on the original vertices. Because hardware tessellation can generate millions of additional triangles, it is essential to minimize the amount of per-vertex computations post-tessellation, and to perform animation calculations only once per control vertex. In the DirectX 11 tessellation pipeline, this can be easily expressed in a single rendering pass using various shader types as described in [Gee08]. In DirectX 9, Xbox 360, and DirectX10/10.1 APIs, a two-pass rendering approach is needed.

In the first pass, we perform all animation calculations for the control cage vertices, and store the results in video memory. This can be implemented using stream output from vertex shaders (DirectX 10) or R2VB and pixel shaders (DirectX 9). In the second pass, tessellation is enabled, and the animated vertices can be directly used by the evaluation shader. In DirectX 10, the multi-pass technique can also be applied to instanced characters, by rendering an instanced point set during the first pass and using the instance and vertex IDs to fetch from a buffer resource during the second pass. Note that using this multi-pass method for control cage rendering is beneficial not only for rendering tessellated characters, but for any rendering pipeline in which we wish to re-use results of expensive vertex operations multiple times. For example, we can use the results of the first pass for our animated and transformed characters for rendering into shadow maps and cube maps for reflections.

Although it is helpful to stream and re-use the animation calculations, this alone is not fully effective. The vertex data will be streamed at full precision, and the evaluation shader must still pay a large cost in memory bandwidth and fetch instructions to retrieve it. To improve performance, we use a compression scheme to pack the transformed vertices into a compact 128-bit format to remove this bottleneck and to reduce the associated memory footprint. This allows the tessellation pass to load a full set of transformed vertex data using a single

fetch per superprimitive vertex. Although the compression scheme requires additional ALU cycles for both compression and decompression, this is more than paid for by the reduction in memory bandwidth and fetch operations in the evaluation shader.

We compress vertex positions by expressing them as fixed-point values which are used to interpolate the corners of a sufficiently large bounding box that is local to each character. The number of bits needed depends on the size of the model and the desired quality level, but it does not need to be extremely large. For example, in the AMD “Froblins” demo, the dynamic range of our vertex data is roughly 600 cm. A 16-bit coordinate on this scale gives a resolution of about 90 microns, which is slightly larger than the diameter of a human hair.

We can compress the tangent frame by converting the basis vectors to spherical coordinates and quantizing them. Spherical coordinates are well suited to compressing unit length vectors, since every compressed value in the spherical domain corresponds to a unique unit-length vector. In a Cartesian representation (such as the widely used `DEC3N` format), a large fraction of the space of compressed values will go unused. What this means in practice is that a much smaller bit count can be used to represent spherical coordinates at a reasonable level of accuracy. We have found that using an 8-bit spherical coordinate pair for tangent space basis vectors results in rendered images that are comparable in quality to a 32-bit Cartesian format. The main drawback of using spherical coordinates is that a number of expensive trigonometric functions must be used for compression and decompression.

Texture coordinates are compressed by converting the *uv* coordinates into a pair of fixed-point values, using whatever bits are left. To ensure acceptable precision, this requires that the *uv* coordinates in the model to the 0-1 range, with no explicit tiling of textures by the artist. For small textures, a smaller bit count could be used for the *uv* coordinates, provided that the *uvs* are snapped to the texel centers.

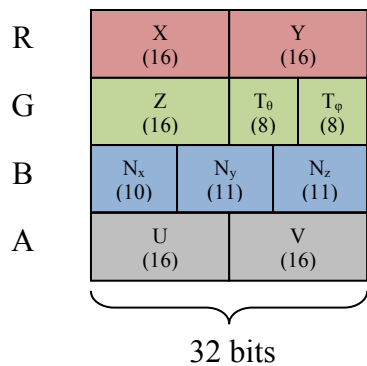


Figure 10. Data format used for compressed, animated vertices

Our bit layout for the compressed vertices is shown in Figure 10, and corresponding compression and decompression code is shown in Listings 10 and 11. We use 16 bits for each component of the position, two 8-bit spherical coordinates for the tangent, 32 bits for the normal, and 16 for each *uv* coordinate. Since our tangent frames are orthogonal, we refrain from storing the binormal, and instead re-compute it based on the decompressed normal and tangent. Since a full 32-bit field is available, we use `DEC3N`-like compression for the normal, which requires fewer ALU operations than spherical coordinates. If additional data fields are needed, we have also found that 8-bit spherical coordinates can be used for the normal, at a quality level comparable to `DEC3N`. We experimented with all of these alternatives on the ATI Radeon HD 4870 GPU, but found little practical difference in performance or quality between any of them.

This compressed format would also make an excellent storage format for static geometry. In this case (also for the case of non-instanced characters), the decompression could be accelerated by leveraging the vertex fetch hardware to perform some of the integer-to-float conversions. We cannot do this for instanced characters,

because we must explicitly fetch vertex data with buffer loads, using the instance ID of the character, instead of using the fixed-function vertex fetch. On average, we obtained a 25% performance improvement via our multi-pass technique with compression (compared to a single-pass method that animates each tessellated vertex).

```
// Quantizes a floating point value (0-1) to a certain number of bits
uint Quantize( float v, uint nBits )
{
    float fMax = ((float) (1 << nBits))-1.0f;
    return uint( round(v*fMax) );
}

uint PackShorts( uint nHigh, uint nLow )
{
    return (nHigh << 16) | (nLow);
}

uint PackBytes( uint nHigh, uint nLow )
{
    return (nHigh << 8) | (nLow);
}

/// Converts a vector to spherical coordinates.
/// Theta (x) is in the 0-PI range. Phi (y) is in the -PI,PI range
float2 CartesianToSpherical( float3 cartesian )
{
    cartesian = clamp( normalize( cartesian ), -1,1 ); // beware of rounding error
    float theta = acos( cartesian.z );
    float s = sqrt( cartesian.x * cartesian.x + cartesian.y * cartesian.y );
    float phi = atan2( cartesian.x / s, cartesian.y / s );
    if( s == 0 )
        phi = 0; // prevent singularity if normal points straight up

    return float2( theta, phi );
}

// Converts a normal vector to quantized spherical coordinates
uint2 CompressVectorQSC( float3 v, uint nBits )
{
    float2 vSpherical = CartesianToSpherical( v );

    return uint2( Quantize( vSphericalNorm.x / PI, nBits ),
                  Quantize( (vSphericalNorm.y + PI) / (2*PI), nBits ) );
}

// Encodes position as fixed-point lerp factors between AABB corners
uint3 CompressPosition( float3 vPos, float3 vBBMin, float3 vBBMax, uint nBits )
{
    float3 vPosNorm = saturate( (vPos - vBBMin) / (vBBMax-vBBMin) );
    return uint3( Quantize( vPosNorm.x, nBits ),
                  Quantize( vPosNorm.y, nBits ),
                  Quantize( vPosNorm.z, nBits ) );
}

uint PackCartesian( float3 v )
{
    float3 vUnsigned = saturate( (v.xyz * 0.5) + 0.5 );
    uint nX = Quantize( vUnsigned.x, 10 );
    uint nY = Quantize( vUnsigned.y, 11 );
    uint nZ = Quantize( vUnsigned.z, 11 );
    return ( nX << 22 ) | ( nY << 11 ) | nZ;
}

uint4 PackVertex( CompressedVertex v, float3 vBBBoxMin, float3 vBBBoxMax )
{
    uint3 nPosition = CompressPosition( v.vPosition, vBBBoxMin, vBBBoxMax, 16 );
    uint2 nTangent = CompressVectorQSC( v.vTangent, 8 );

    uint4 nOutput;
    nOutput.x = PackShorts( nPosition.x, nPosition.y );
    nOutput.y = PackShorts( nPosition.z, PackBytes( nTangent.x, nTangent.y ) );
    nOutput.z = PackCartesian( v.vNormal );
    nOutput.w = PackShorts( Quantize( vUV.x, 16 ), Quantize( vUV.y, 16 ) );
    return nOutput;
}
```

Code Listing 11. Compression code for vertex format given in Figure 10.

```

float DeQuantize( uint n, uint nBits )
{
    float fMax = ((float) (1 << nBits)) - 1.0f;
    return float(n)/fMax;
}

float3 DecompressVectorQSC( uint2 nCompressed, uint nBitCount )
{
    float2 vSph = float2( DeQuantize( nCompressed.x, nBitCount ),
                          DeQuantize( nCompressed.y, nBitCount ) );
    vSph.x = vSph.x * PI;
    vSph.y = (2 * PI * vSph.y) - PI
    float fSinTheta = sin( vSph.x );
    float fCosTheta = cos( vSph.x );
    float fSinPhi = sin( vSph.y );
    float fCosPhi = cos( vSph.y );
    return float3( fSinPhi * fSinTheta, fCosPhi * fSinTheta, fCosTheta );
}

float3 DecompressPosition( uint3 nBits, float3 vBBMin, float3 vBBMax, uint nCount )
{
    float3 vPosN = float3( DeQuantize( nBits.x, nCount),
                          DeQuantize( nBits.y, nCount),
                          DeQuantize( nBits.z, nCount) );
    return lerp( vBBMin.xyz, vBBMax.xyz, vPosN );
}

float3 UnpackPosition( uint4 nPacked, float3 vBBBoxMin, float3 vBBBoxMax )
{
    uint3 nPos;
    nPos.xy = uint2( nPacked.x >> 16, nPacked.x & 0x0000ffff );
    nPos.z = nPacked.y >> 16;
    return DecompressPosition( nPos, vBBBoxMin, vBBBoxMax, 16 );
}

float2 UnpackUV( uint4 nPacked )
{
    uint2 nUV = uint2( nPacked.w >> 16, nPacked.w & 0x0000ffff );
    float2 vUV = float2( DeQuantize( nUV.x, 16 ), DeQuantize( nUV.y, 16 ) );
    return vUV;
}

float3 UnpackTangent( uint4 nPacked )
{
    uint2 nTan = uint2( (nPacked.y >> 8) & 0xff, nPacked.y & 0xff );
    return DecompressVectorQSC( nTan, 8 );
}

float3 UnpackCartesian( uint n )
{
    uint nX = (n >> 22) & 0x3FF;
    uint nY = (n >> 11) & 0x7FF;
    uint nZ = n & 0x7FF;
    float fX = (2.0f * DeQuantize( nX, 10 )) - 1.0f;
    float fY = (2.0f * DeQuantize( nY, 11 )) - 1.0f;
    float fZ = (2.0f * DeQuantize( nZ, 11 )) - 1.0f;
    return float3( fX, fY, fZ );
}

CompressedVertex UnpackVertex( uint4 nPacked, float3 vBBBoxMin, float3 vBBBoxMax )
{
    CompressedVertex vVert;
    vVert.vPosition = UnpackPosition( nPacked, vBBBoxMin, vBBBoxMax );
    vVert.vNormal = UnpackCartesian( nPacked.z );
    vVert.vTangent = UnpackTangent( nPacked );
    vVert.vBinormal = normalize( cross( vVert.vTangent, vVert.vNormal ) );
    vVert.vUV = UnpackUV( nPacked );
    return vVert;
}

```

Code Listing 12. Decompression code for vertex format given in Figure 10.

Adaptive Tessellation Mode

By using the adaptive tessellation mode, we can specify a tessellation level for each individual edge of the control mesh primitives (superprims). This allows us to *modify* the level of detail for the mesh at render time, using any number of metrics, in a non-uniform fashion. For example, one LOD metric can target a specific screen-space edge length to maintain a reasonable triangle size independent of the object's position with respect to the camera. Thus we wouldn't waste performance on details that would be unnoticed in the final rendering. The actual resolution of the screen would be automatically accounted for with this LOD technique as well. Alternatively, we can use a vertex-to-camera distance metric, or a silhouette-based metric to adaptively subdivide the mesh (in the latter case we would subdivide more at silhouettes to preserve them). This mode provides stable performance by allowing the triangle density to be adjusted locally, so that triangles are created only where needed.

Figure 12 shows some examples of superprimitive tessellation using adaptive tessellation. With adaptive tessellation we can also specify a maximum and a minimum tessellation level per draw call. These will serve as range clamps for the individual edge levels. With Direct3D 9 and the Xbox 360, we specify the per-edge tessellation levels via a separate vertex stream along with the regular mesh data (the original vertex stream). All adaptive draw calls in Direct3D 9 and Xbox 360 must be rendered with non-indexed draw calls. With Direct3D 10/10.1, the tessellation factors are specified directly via the tessellation library interface. Note that, to implement dynamic adaptive tessellation, we need to be able to compute per-edge tessellation factors per frame. This is typically done in a pass prior to the tessellation pass at render time.

One immediate concern with using non-indexed rendering for adaptive tessellation draw calls is that typically we render with indexed meshes to optimize for post-transform cache coherence. However, since the tessellator does not generate duplicate vertices, and the original control cage meshes are rather small to start with, using non-indexed meshes does not impose significant performance penalties.

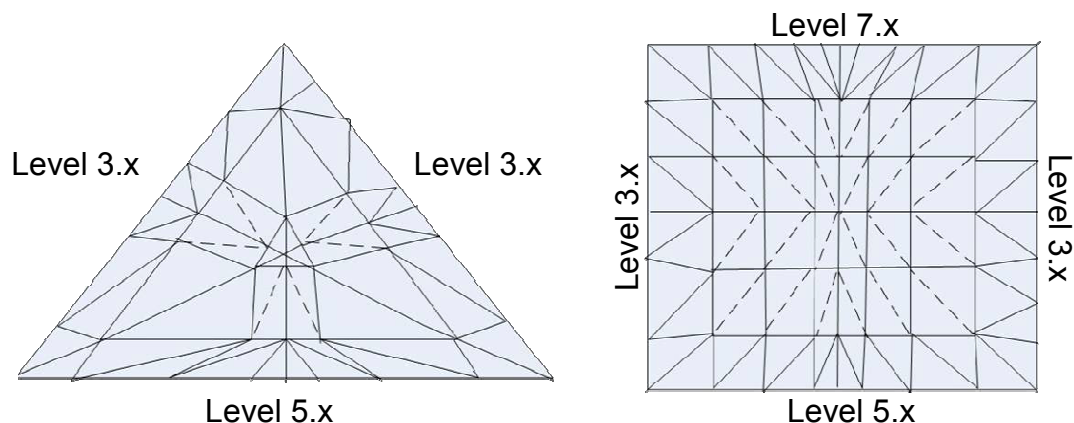


Figure 12. Examples of adaptive tessellation for triangle and quad primitives.

Programming for Adaptive GPU Tessellation

To render with adaptive tessellation, we need first to compute the tessellation factors for all edges in the original mesh. Then we can render our mesh with the associated tessellation factors. In Direct3D 9 we can compute the tessellation factors by using the R2VB extension [PERSSON06], which allows us to re-circulate vertex data as render targets and vertex buffers as needed.

We can use a number of level-of-detail metrics to control the quality of rendered terrain and adaptively subdivide to select the right amount of detail as needed. One approach is to dynamically compute tessellation factors in a pre-pass to determine how finely to tessellate each polygon in the terrain. We can calculate the dynamic level of detail per primitive edge on the GPU using a metric for maintaining near-constant screen-space edge length. In this case, the LOD metric can target a specific screen-space edge length to maintain a reasonable triangle size independent of the object's position with respect to the camera; thus, we would not waste performance on details that would go unnoticed in the final rendering. The actual resolution of the screen would be automatically accounted for with this LOD technique as well. Alternatively (or in addition to this metric), we can also account for the details in the displacement map and tessellate more finely around the areas of high displacement. We can also use a vertex-to-camera distance metric or a silhouette-based metric to adaptively subdivide the mesh (in the latter case, we would subdivide more at silhouettes to preserve them). This mode provides stable performance by allowing local adjustment of the triangle density, thus creating triangles are created only where needed.

As an example, the following is a description of a DirectX 9 multi-pass method for computing the tessellation factors based on distance from the camera:

Since DirectX 9 requires a non-indexed format for the vertex buffer, we need to copy the original low res mesh super-prim indexed vertex data into a new non-indexed vertex buffer. While copying the data, a unique vertex ID is assigned to each vertex and stored in the w component of the vertex position data. This vertex ID is used by the vertex shader when calculating tessellation factors to associate a tessellation factor with an edge in the mesh. For each triangle in the mesh, the vertex IDs are ordered consecutively so that a triangle is formed by vertex IDs: n , $n+1$, and $n+2$.

Figure 13 provides an overview of the passes for rendering with adaptive tessellation. The first pass for calculating tessellation factors renders the new non-indexed low res mesh as point primitives with the tessellator disabled. This pass is used to transform the vertex data into view space to make calculation of the distance to the camera simpler. The vertex shader transforms the location of each point primitive so that they are stored contiguously in a 2D array based on the order of the vertex ID. The view space location of the point primitives are then stored as the color values which are saved as pixels in the render target.

The second pass uses the render target from the first pass as a vertex texture. The render target for this pass will be an R2VB render target which will allow it to be use as a vertex buffer in a subsequent pass. For details on using R2VB please read the AMD technical report "Render to Vertex Buffer Programming" [Persson06]. As with the first pass, this pass will render the non-indexed low res mesh as point primitives. This time, the vertex shader will be used for calculating the actual per-edge tessellation factors. In the vertex shader, the current vertex is fetched from the texture containing the view space transformed vertices. The vertex ID is used to calculate the texture coordinates for the transformed vertex. The next vertex in the current triangle is also fetched. This gives us the two vertices that make up the current triangle edge. At this point we can now calculate the distance from the camera to the midpoint on the edge. Since the vertices are in view space, the calculation just needs the z values of the two vertices. The vertex shader can then use this value to compute the tessellation factor for the edge by converting it to a number between the minimum and maximum allowable tessellation factors. Note that larger distances should correspond to smaller tessellation factors since we want less detail with more distant edges.

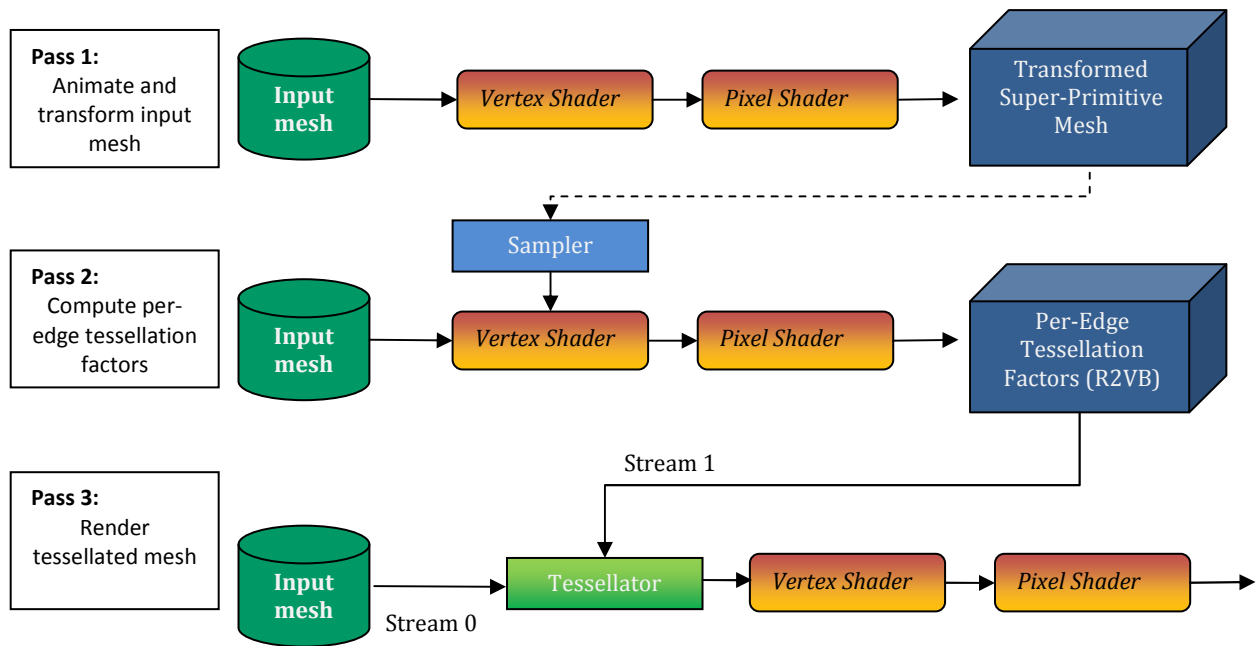


Figure 13. Visual representation of adaptive tessellation rendering passes.

Transforming Mesh Using R2VB

In the first pass, we need to render our input mesh to compute its transformed and animated version. This, in essence, emulates the data re-circulation functionality of flexible buffer support available in Direct3D 10.

In this pass, we render our input mesh as point list primitives. The vertex shader transforms the primitives and outputs them to a 2D buffer. We transform the vertices in this pass to screen space, because our LOD metric for computing the per-edge tessellation factors is based on maintaining near-constant screen-space edge length for the triangles in the rendered mesh. Note that since we are using displacement mapping for our terrain, all displacement mapping and animation must happen prior to transforming the vertices into screen space. The vertex shader computes the 2D coordinates and the pixel shader renders the results into the appropriate render target. In our case, this render target is a texture used to store the transformed mesh.

```

IDirect3DTexture9* sg_pTransformedVerts = NULL;

// Create the texture for the transformed super-prim texture
V( pd3dDevice->CreateTexture( cg nR2VBTextureWidth, sg nR2VBTextureHeight, 1,
    D3DUSAGE_RENDERTARGET, D3DFMT_A32B32G32R32F,
    D3DPOOL_DEFAULT, &sg_pTransformedVerts, NULL ));

// Create the R2VB handle for the tessellation factors
V( r2vbInit( pd3dDevice, nBufferSize * sizeof(float), &sg hR2VB TessFactors ));

...
IDirect3DSurface9* pCurrentRT = NULL; // the backbuffer or currently rendered to surface
IDirect3DSurface9* pNewRT      = NULL;

// Get a pointer to the backbuffer:
V( pd3dDevice->GetRenderTarget( 0, &pCurrentRT ));

// Select whether we're applying displacement or not:
if ( sg_bDisplaced )
    sg_pEffect->SetTechnique( "RenderTransformedDisplacedVerts" );
else
    sg_pEffect->SetTechnique( "RenderTransformedVerts" );

// Set the vertex declaration and stream source to the original low-res input mesh:
V( pd3dDevice->SetVertexDeclaration( sg_pVertexIDDecl ));
V( pd3dDevice->SetStreamSource( 0, sg_pNonIndexedMeshVB, 0, nVertexSize ));

```

```

// Output transformed vertices for the input low-res mesh to a render target for use in
// the next pass:
V( sg_pTransformedVerts->GetSurfaceLevel( 0, &pNewRT ));
V( pd3dDevice->SetRenderTarget( 0, pNewRT ));

// Render the passes to transform the vertices:
V( sg_pEffect->Begin( &cPasses, 0 ));
for ( iPass = 0; iPass < cPasses; iPass++ )
{
    V( sg_pEffect->BeginPass( iPass ));

    // Render as points and draw to an R2VB buffer:
    V( pd3dDevice->DrawPrimitive( D3DPT_POINTLIST, 0, sg_nNumVerts ));

    V( sg_pEffect->EndPass( ));
}
V( sg_pEffect->End( ));

SAFE_RELEASE( pNewRT );

```

Code Listing 13. Example Direct3D code for setting up the first pass for rendering transformed vertices into an R2VB render target.

In a Direct3D 9 implementation we need to pre-compute vertex IDs and store them in the input super-primitive mesh. Unfortunately, vertex IDs are not provided by the run-time in Direct3D 9 (unlike Direct3D 10 and beyond). These vertex IDs are necessary for calculation of correct 2D texture coordinates to output the transformed vertex to the render target (for the transformed mesh). We also need to propagate that the ID value to the next pass because the vertex IDs are also needed to accurately calculate per-edge tessellation factors. Therefore, we simply pre-compute these values at mesh preprocessing time and store them as the fourth component of the input mesh's position element.

```

struct VsTfmInput
{
    float4 vPosOS    : POSITION0;    // (x,y,z) - 3D position, w - vertex ID (precomputed)
    float2 vTexCoord : TEXCOORD0;
    float3 vNormal   : NORMAL0;
};

struct VsTfmOutput
{
    float4 vPos      : POSITION;
    float  fVertexID : TEXCOORD0;
    float3 vPosCS    : TEXCOORD1;
};

VsTfmOutput VSRenderTransformedDisplaced( VsTfmInput i )
{
    VsTfmOutput o;
    int nVertexID    = floor( i.vPosOS.w );
    int nTextureWidth = g_vTformVertsMapSize.x;

    // Compute row and column of the position in 2D texture
    float2 vPos      = float2( nVertexID % nTextureWidth, nVertexID / nTextureWidth );
    vPos /= g_vTformVertsMapSize.xy;
    vPos.y = 1.0 - vPos.y;
    vPos    = vPos * 2 - 1.0; // Move to [-1; 1] range

    o.vPos = float4( vPos.xy, 0, 1 );

    // Propagate the vertex ID to the pixel shader
    o.fVertexID = i.vPosOS.w;

    // Displace input vertex:
    float4 vPositionCS = float4( DisplaceVertex( i.vPosOS, i.vTexCoord, i.vNormal ), 1 );

    // Transform vertex position to screen-space
    vPositionCS = mul( vPositionCS, g_mWorldView );
    vPositionCS /= vPositionCS.w;

    o.vPosCS = vPositionCS.xyz;
}

```

```

        return 0;

    } // End of VsTfmOutput VSRenderTransformedDisplaced(..)

    //-----
    struct PsTfmInput
    {
        float4 vPos      : POSITION;
        float  fVertexID  : TEXCOORD0;
        float3 vPosCS     : TEXCOORD1;
    };

    float4 PSRenderTransformed( PsTfmInput i ): COLOR0
    {
        return float4( i.vPosCS, i.fVertexID );
    }

    //-----
    technique RenderTransformedDisplacedVerts
    {
        pass P0
        {
            CullMode      = NONE;
            ZEnable        = false;
            ZWriteEnable   = false;

            VertexShader   = compile vs_3_0 VSRenderTransformedDisplaced();
            PixelShader     = compile ps_3_0 PSRenderTransformed();
        }
    } // End of technique RenderTransformedVerts

```

Code Listing 14. Example D3DX FX shader for rendering transformed vertices as points into an R2VB render target. Note that some methods (such as *DisplaceVertex(..)* are omitted for brevity, but are included in the accompanying SDK sample.

Computing Per-Edge Tessellation Factors

In the next pass, we compute the factors that determine how finely each edge of a given primitive will be tessellated during the tessellated mesh draw call. This is the heart of adaptive subdivision and LOD management for terrain rendering. There are many different ways to compute these per-edge factors using a number of metrics, as we have mentioned. The important consideration is smoothness of LOD transition (as the vertices change their relationship to the viewer, we should not see sudden pops in lighting and vertex positions). This is particularly important given that, because we are using displacement mapping, sudden changes in terrain silhouettes can be very apparent.

For this pass, we also render our input mesh as a point list. We will use the previously computed transformed vertices resource and bind it as a vertex sampler to this pass. The vertex shader will rely on the provided vertex ID to compute tessellation factors for each edge and to sample the resource to fetch the transformed vertex data. Since we are rendering with a non-indexed mesh, we can compute one tessellation edge factor per input vertex and output it to a 2D R2VB render target. Note that the setup in Direct3D 9 is very similar to pass 1, and we refer the reader to the sample SDK application on the DVD for details.

We need to ensure that our resulting surface does not display cracks when applying displacement during rendering. Hence, the adjacent edges must have precision-identical tessellation factors; otherwise, we will instantly notice cracks in the terrain once displacement is applied (Figure 14). Furthermore, extra care during tessellation factor computation needs to be taken to ensure identical edge direction when evaluating the surface. We use vertex ID for this purpose. See Listing 15 for the details of our approach.

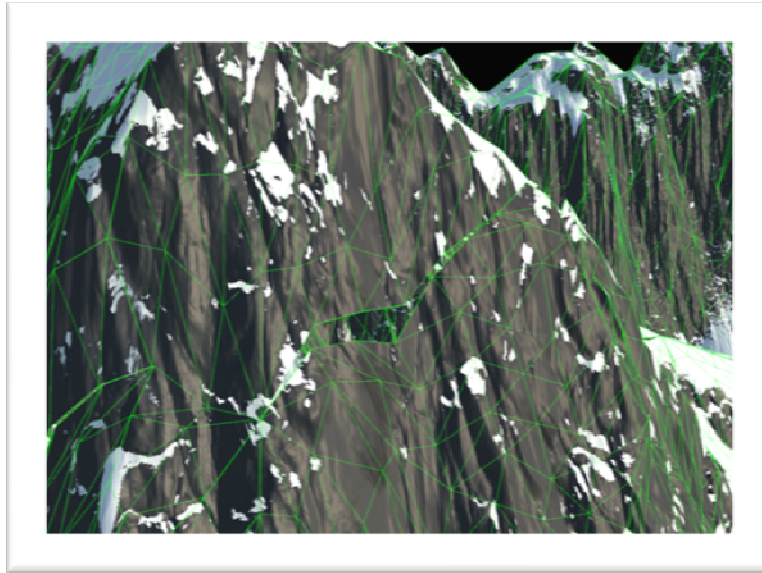


Figure 14. Example of cracks in the terrain surface when computing tessellation factors for edges without aligning edge direction.

```

struct VsTFInput
{
    float4 vPositionOS : POSITION0; // (xyz) - camera-space position, w - vertex ID
};

struct VsTFOutput
{
    float4 vPositionCS : POSITION;
    float fTessFactor : TEXCOORD0;
};

//-----
VsTFOutput VSCalculateTFs( VsTFInput i )
{
    VsTFOutput o;

    // Current vertex ID:
    int nCurrentVertID = (int)( i.vPositionOS.w );

    // Determine the ID of the edge neighbor's vertex (remember that this is done with
    // non-indexed primitives, so basically if the current vertex is v0 ,then we have
    // edges: v0->v1, v1->v2, v2->v0

    int nCurrentVertEdgeID = nCurrentVertID - 3 * floor( (float)nCurrentVertID /
                                                         (float) 3);

    int nEdgeVert0ID = nCurrentVertID; // this works if current vertex is v0 or v1
    int nEdgeVert1ID = nCurrentVertID + 1;

    if ( nCurrentVertEdgeID == 0 )
    {
        nEdgeVert0ID = nCurrentVertID + 1;
    }
    else if ( nCurrentVertEdgeID == 1 )
    {
        nEdgeVert0ID = nCurrentVertID + 1;
    }
    else if ( nCurrentVertEdgeID == 2 ) // In case of v2 we need to wrap around to v0
    {
        nEdgeVert0ID = nCurrentVertID - 2;
    }

    // Compute the fetch coordinates to fetch transformed positions
    // for these two vertices to compute their edge statistics:
    int nTextureWidth = g_vTformVertsMapSize.x;

    // Vertex0: nCurrentVertID, compute row and column of the position in 2D texture:
    float2 vVert0Coords = float2( nCurrentVertID % nTextureWidth,
                                   nCurrentVertID / nTextureWidth );
    vVert0Coords /= g_vTformVertsMapSize.xy;

```

```

// Vertex1: nEdgeVert0ID, compute row and column of the position in 2D texture:
float2 vVert1Coords = float2( nEdgeVert0ID % nTextureWidth,
                               nEdgeVert0ID / nTextureWidth );
vVert1Coords /= g_vTformVertsMapSize.xy;

// Fetch transformed positions for these IDs:
float4 vVert0Pos = tex2Dlod( sTformVerts, float4( vVert0Coords, 0, 0 ) );
float4 vVert1Pos = tex2Dlod( sTformVerts, float4( vVert1Coords, 0, 0 ) );

// Swap vertices to make sure that we have the same edge direction
// regardless of their triangle order (based on vertex ID):
if ( vVert0Pos.w > vVert1Pos.w )
{
    float4 vTmpVert = vVert0Pos;
    vVert0Pos = vVert1Pos;
    vVert1Pos = vTmpVert;
}

// Use the distance from the camera to determine the tessellation factor:
float fEdgeDepth = 0.5 * ( ( vVert1Pos.z ) + ( vVert0Pos.z ) );

float fTessFactor = clamp( fEdgeDepth / g_fMaxCameraDistance,
                           0, 1 ); // Map to 0-1 range
fTessFactor = (1 - fTessFactor) * 15; // map to 0-15 range and invert it
// (higher to lower)

const float fMinTessFactor = 1.0;
const float fMaxTessFactor = 14.99999;

// Clamp to the correct range
o.fTessFactor = clamp( fTessFactor, fMinTessFactor, fMaxTessFactor );

// Compute output position for rendering into a tessellation factors texture
int nVertexID = floor( i.vPositionOS.w );

// Compute row and column of the position in 2D texture
float2 vOutputPos = float2( nVertexID % nTextureWidth, nVertexID / nTextureWidth );
vOutputPos /= g_vTformVertsMapSize.xy;
vOutputPos.y = 1.0 - vOutputPos.y;
vOutputPos = vOutputPos * 2 - 1.0; // Move to [-1; 1] range

o.vPositionCS = float4( vOutputPos.xy, 0, 1 );

return o;
} // End of VsTfmOutput VSCalculateTFs(..)

// Pixel shader for rendering the tessellation factor into an R2VB buffer.
struct PsTFInput
{
    float fTessFactor : TEXCOORD0;
};

//-----
float4 PSCalculateTFs( PsTFInput i ): COLOR0
{
    return i.fTessFactor.xxxx;
}

//.....
technique CalculateTessellationFactors
{
    pass P0
    {
        CullMode      = NONE;
        ZEnable       = false;
        ZWriteEnable   = false;

        VertexShader  = compile vs_3_0 VSCalculateTFs();
        PixelShader    = compile ps_3_0 PSCalculateTFs();
    }
} // End of technique CalculateTessellationFactors

```

Code Listing 15. Example HLSL code for computing per-edge tessellation factors based on screen-space edge length.

Rendering Tessellated Mesh with Per-Edge Tessellation Factors

In the next pass, we will use the newly computed tessellation factors to render the tessellated and displaced mesh. We will bind the input mesh as input vertex stream 0 and will bind the tessellation factors as stream 1 (see Listing 16). The underlying driver and hardware platform will automatically use these tessellation factors to generate new surface locations and propagate them to the tessellation evaluation vertex shader.

```
// Enable adaptive tessellation (specifying that tessellation factors are given in vertex
// stream 1 - 3rd parameter):
sg_pTessInterface->SetMode( TSMD_ENABLE_ADAPTIVE, 1 );

// Set min tessellation level:
sg_pTessInterface->SetMinLevel( IATITessellationD3D9::ATI_TESS_MIN_TESSELLATION_LEVEL );

// Set tessellation level (this is used for both continuous and adaptive:
sg_pTessInterface->SetMaxLevel( sg_fMaxTessellationLevel );

// Set the vertex declaration and stream source for adaptively tessellated mesh:
V( pd3dDevice->SetVertexDeclaration( sg_pAdaptiveVBDecl ));
V( pd3dDevice->SetStreamSource( 0, sg_pNonIndexedMeshVB, 0, nVertexSize ));

// Map the tessellation factors R2VB to stream source 1:
V( r2vbSetStreamTexSource( sg_hr2VBTEssFactors, 1, 0, sg_pTessFactors, 0, 4 ));

// Select the appropriate technique to render our tessellated objects:
//
if ( sg_bDisplaced )
    if ( sg_bWireframe )
        sg_pEffect->SetTechnique( "RenderTessellatedDisplacedSceneWireframe" );
    else
        sg_pEffect->SetTechnique( "RenderTessellatedDisplacedScene" );
else
    if ( sg_bWireframe )
        sg_pEffect->SetTechnique( "RenderTessellatedSceneWireframe" );
    else
        sg_pEffect->SetTechnique( "RenderTessellatedScene" );

V( sg_pEffect->Begin( &cPasses, 0 ));
for ( iPass = 0; iPass < cPasses; iPass++ )
{
    V( sg_pEffect->BeginPass( iPass ));
    V( sg_pTessInterface->DrawNonIndexed( TSPT_TRIANGLELIST, 0, sg_nNumVerts / 3 ));
    V( sg_pEffect->EndPass( ));
}
V( sg_pEffect->End( ));

// Disengage R2VB tessellation factors:
r2vbDisTexSource( sg_hr2VBTEssFactors, 0 );

// Disable tessellation:
sg_pTessInterface->SetMode( TSMD_DISABLE );
```

Code Listing 16. Example of Direct3D 9 and ATI GPU tessellation library calls for rendering adaptively tessellated terrain.

```
//-----
struct VsInputTessellated
{
    // Barycentric weights for tessellated vertex from the super-primitive (low-resolution
    // input) mesh (i.e., each tessellated vertex quantities are computed via barycentric
    // interpolation from the three super-primitive input vertices)
    float3 vBarycentric: BLENDWEIGHT0;

    // Data from super-prim vertex 0:
    float4 vPositionVert0 : POSITION0;
    float2 vTexCoordVert0 : TEXCOORD0;
    float3 vNormalVert0   : NORMAL0;

    // Data from super-prim vertex 1:
    float4 vPositionVert1 : POSITION4;
    float2 vTexCoordVert1 : TEXCOORD4;
    float3 vNormalVert1   : NORMAL4;
```

```

// Data from super-prim vertex 2:
float4 vPositionVert2 : POSITION8;
float2 vTexCoordVert2 : TEXCOORD8;
float3 vNormalVert2   : NORMAL8;
};

struct VsOutputTessellated
{
    float4 vPosCS      : POSITION;
    float2 vTexCoord    : TEXCOORD0;
    float3 vNormalWS    : TEXCOORD1;
    float3 vPositionWS  : TEXCOORD2;
};
//-----
// Render tessellated mesh with displacement
//-----
VsOutputTessellated VSRenderTessellatedDisplaced( VsInputTessellated i )
{
    VsOutputTessellated o;

    // Compute new position based on the barycentric coordinates:
    float3 vPosTessOS = i.vPositionVert0.xyz * i.vBarycentric.x +
                        i.vPositionVert1.xyz * i.vBarycentric.y +
                        i.vPositionVert2.xyz * i.vBarycentric.z;

    // Output world-space position:
    o.vPositionWS = vPosTessOS;

    // Compute new tangent space basis vectors for the tessellated vertex:
    o.vNormalWS    = i.vNormalVert0.xyz * i.vBarycentric.x +
                    i.vNormalVert1.xyz * i.vBarycentric.y +
                    i.vNormalVert2.xyz * i.vBarycentric.z;

    // Compute new texture coordinates based on the barycentric coordinates:
    o.vTexCoord = i.vTexCoordVert0.xy * i.vBarycentric.x +
                 i.vTexCoordVert1.xy * i.vBarycentric.y +
                 i.vTexCoordVert2.xy * i.vBarycentric.z;

    // Displace the tessellated vertex:
    o.vPositionWS = DisplaceVertex( vPosTessOS, o.vTexCoord, o.vNormalWS );

    // Transform position to screen-space:
    o.vPosCS = mul( float4( o.vPositionWS, 1.0 ), g_mWorldViewProjection );

    return o;
} // End of VsOutputTessellated VSRenderTessellatedDisplaced(..)

```

Code Listing 17. *Example vertex shader for rendering adaptively tessellated terrain.*

Shading Tessellated Displaced Surfaces

We shade the terrain object using the height map for lighting information (Figure 15) without relying on terrain normal maps. This allows us to reduce the overall memory footprint for this subsystem. The height map was designed using World-Machine terrain modeling software [WorldMachine08], which can generate very large and complex terrain maps. We use a 2K x 2K height map for our terrain object, and this can be extended to higher spans by combining multiple patches or by using larger map sizes (for example, by converting to Direct3D 10).

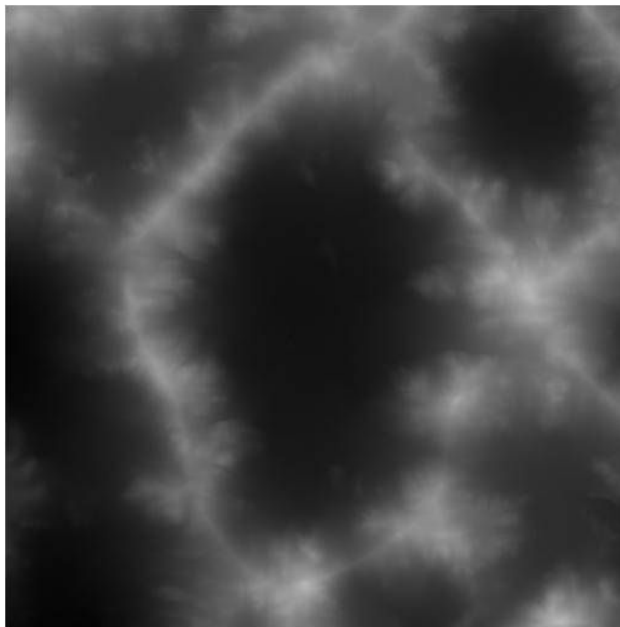


Figure 15. An example height map used to render our terrain object.

We dynamically derive the normal for each rendered pixel from the height map itself (see Listing 18) using central differences approximation on the height field. There are a number of ways to implement this, and our approach is simply one such method. Another method is outlined in [AnderssonTatarchuk07].

```

texture  sMap;          // Displacement map
float    fScale;        // Displacement scale
float    fBias;         // Displacement bias

//-----
float DisplacementTapMipped( float2 vUV, sampler2D sMap, float fScale, float fBias )
{
    float fDisplacement = tex2D( sMap, float4( vUV, 0, 0 ) ).r;
    fDisplacement = fDisplacement * fScale + fBias;

    return fDisplacement;
}

//-----
/// Calculates a displaced tangent frame at a particular point in a displacement map.
/// The displacement map is unsigned (0-1 maps to 0-fDisplaceScale).
/// This function returns the modified tangent frame, expressed in the local tangent
/// space. This function uses a tangent space in which Y is up.
///
/// \param tDMap          The displacement map to sample
/// \param sSampler        The sampler to sample with
/// \param fDisplaceScale  Scale factor for converting from normalized displacement
///                          map space into world space
/// \param fDu             Distance traveled in object space per unit change in U.
/// \param fDv             Distance traveled in object space per unit change in V.
/// \param vUV            Coordinates at which to sample
/// \param vNormalTS       The displaced normal, in local tangent space.
/// \param vTangentTS      The displaced tangent, in local tangent space.
/// \param vBinormalTS     The displaced binormal, in local tangent space.
//-----
void CalculateTangentFrameUnsigned( Texture2D<float> tDMap,
                                   sampler          sSampler,
                                   float             fDisplaceScale,
                                   float             fDu,
                                   float             fDv,
                                   float2            vUV,
                                   out float3         vNormalTS,
                                   out float3         vTangentTS,
                                   out float3         vBinormalTS )
{
    float2 vDMapSize;
    tDMap.GetDimensions( vDMapSize.x, vDMapSize.y );

    float fDistanceU = fDu/vDMapSize.x;
    float fDistanceV = fDv/vDMapSize.y;

```

```

float fNeighborU = fDisplaceScale *
    tDMap.SampleLevel( sSampler, vUV + float2(1, 0)/vDMapSize.x,0);
float fNeighborV = fDisplaceScale *
    tDMap.SampleLevel( sSampler, vUV + float2(0, 1)/vDMapSize.y,0);
float fNeighborU2 = fDisplaceScale *
    tDMap.SampleLevel( sSampler, vUV + float2(-1,0)/vDMapSize.x,0);
float fNeighborV2 = fDisplaceScale *
    tDMap.SampleLevel( sSampler, vUV + float2(0,-1)/vDMapSize.y,0);

// Compute normal by central differencing
//
//      V2
//    U2 X  U
//      V
//
//    --> X
//      |
//    V  Z
//
float fDhDu = (fNeighborU - fNeighborU2);
float fDhDv = (fNeighborV - fNeighborV2);

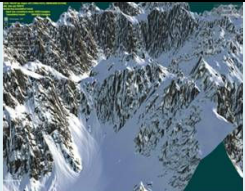
vTangentTS = normalize( float3( fDistanceU, fDhDu/2, 0 ) );
vBinormalTS = normalize( float3( 0, fDhDv/2, fDistanceV ) );
vNormalTS = cross( vBinormalTS, vTangentTS );
}

```

Code Listing 18. An example method for computing the normal from the height map using central differences approach.

Aside from reducing the associated memory footprint, this method provides easy support for dynamically combining multiple displacement maps. This can be very beneficial for interactive modifications of terrain, via destruction or other means. For example, if we were to try to represent a terrain that gets affected by a meteorite shower, we could represent the terrain height field as a dynamic render target and render (splat) the meteorite crater height fields in the regions of impact on top of the main terrain height field. Then we would simply tessellate our terrain quad using this height field and shade by deriving the normal from this combined height field.

Performance Analysis

Rendering Mode	Polygon Count	 			
		Rendering Time			
		ATI Radeon HD 4870	ATI Radeon HD 2900 XT	ATI Radeon HD 4870	ATI Radeon HD 2900 XT
Original low-resolution input mesh (N_L)	4K triangles	852 fps	359 fps	850 fps	275 fps
Tessellated mesh, continuous tessellation (N_T) ¹	1.6M triangles	232 fps	143 fps	213 fps	101 fps

¹ The relationship between the polygonal count of the input mesh and the continuously tessellated mesh can be expressed as $N_T = 411 \times N_L$ on Direct3D 10 generation hardware

Tessellated mesh, adaptive tessellation (N_A)	Dynamic, $N_T \geq N_A$ $\geq N_L$	845 fps	356 fps	574 fps	207 fps
---	--	---------	---------	---------	---------

Table 2. Performance data for comparing rendering terrain patch with different rendering modes. Note that rendering terrain with adaptive tessellation results in much higher quality visual results while maintaining rendering times close to the original, low-resolution mesh rendering.

We measured performance for our application on a quad-core AMD Phenom™ X4 processor-based system with 2GB of RAM and an ATI Radeon graphics cards. Table 2 provides a detailed overview of the results. We notice that adaptive tessellation renders at frame rates close to the original low-resolution rates (less than 1% difference). In the case of rendering the original low-resolution input mesh as well as with rendering using continuous tessellation, we render the mesh as indexed primitives whereas, with adaptive tessellation, we use non-indexed primitives. As mentioned earlier in this chapter, given the results of using adaptive tessellation, we notice the lack of any serious performance penalties due to rendering with non-indexed primitives in the adaptive tessellation case. Instead, the performance is easily comparable to the cost of rendering the original low-resolution mesh, even though the resulting rendered terrain surface is significantly higher quality when rendered with adaptive tessellation. Even by simply turning on adaptive tessellation in far-away views, we double the polygon count for the rendered mesh.

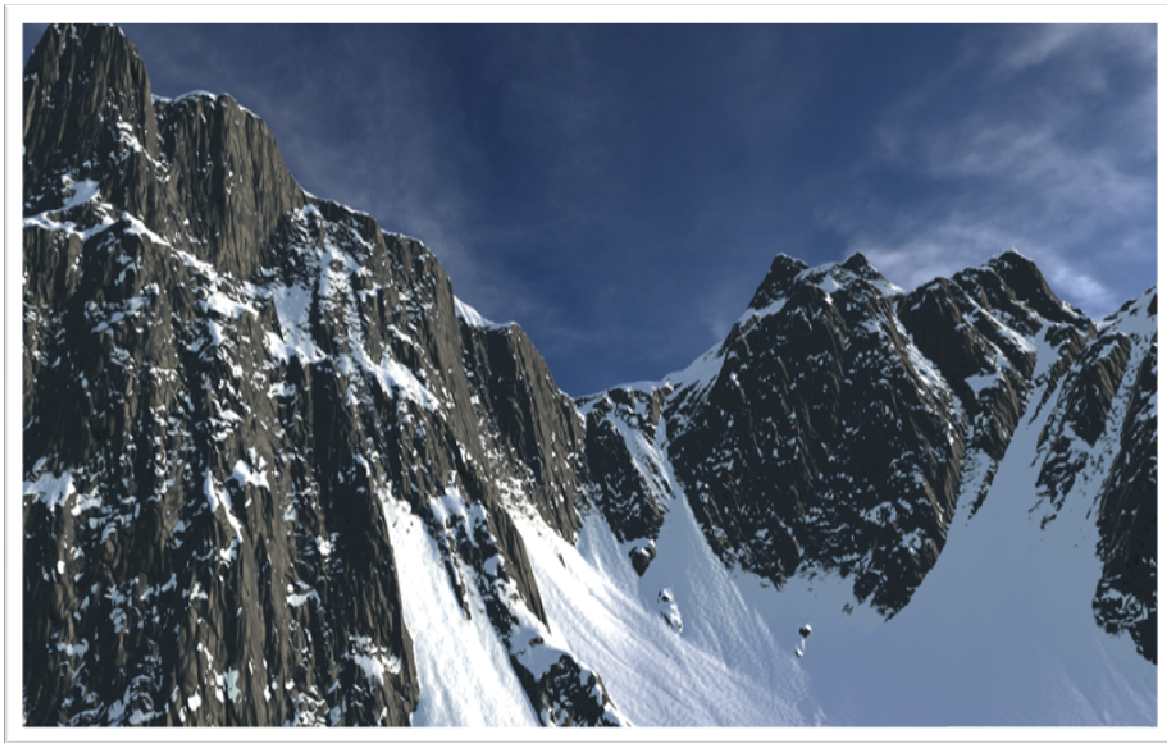


Figure 16. Real-time rendering of terrain with displacement map-based lighting and procedurally generated snow placement.

Displacement Map Tips and Reducing Surface Cracks

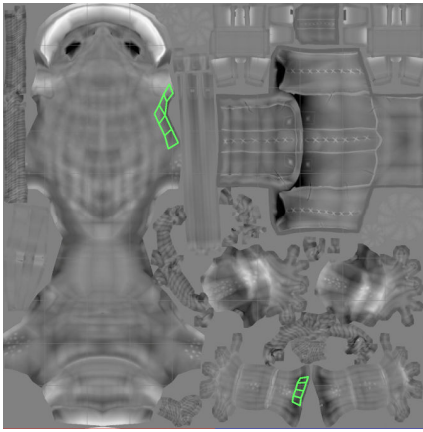
In this section, we share several practical tips for generation and use of displacement maps that we have collected throughout our process.

The method used for generation of displacement maps must match the method for evaluating the subdivided surface. This naturally correlates to the absolute need to know the process used by the modeling tool used for map generations. Many DCC tools (for example, Autodesk Maya) will first use an approximating subdivision process, such as Catmull-Clark subdivision method, on the control mesh (the low-resolution, or superprimitive, mesh). Once the mesh has been smoothed, then the fine-scale details are captured into a scalar or vector displacement map. If we wish to capture the displacement map with this method, and use it during rendering, we must render the final surface using Catmull-Clark subdivision methods during surface evaluation. Combining displacement maps generated with a different surface evaluation method than used for rendering can lead to unpredictable results. Additionally, a number of concerns arise with topology choices and the treatment of extraordinary points, as well as patch reordering to ensure watertightness during displacement. Some interesting solutions have been described in [Castaño08].

In our case, we used the AMD GPUMeshMapper tool [AMDGMM08], designed specifically for robust generation of displacement maps for interpolative planar subdivision. Given a pair of low- and high-resolution meshes, this tool provides a number of different options for controlling the envelopes for ray casting between the low- and high-resolution models to capture displacement and normal information. Furthermore, to achieve controllable results at run-time, we must know the exact floating point values for displacement scale and bias for the generated displacement map. This tool provides this information, collected during the generation process, in the form of parameters that can be used directly in the shader.

Particular care needs to be taken during displacement mapping to generate watertight, or simply visibly crack-free, surfaces. This is true regardless of the subdivision method used for evaluation. One challenge with rendering complex characters with displacement maps that contain texture uv borders is the introduction of texture uv seams (see Figure 17, left, for an example of such a displacement map). When sampling across the uv seams, bilinear discontinuities, along with potentially varying floating point precision of the uv values, may introduce geometry cracks (Figure 17, center) across these uv borders. One solution to resolve this issue is to lay out the neighboring uv borders with the exact sample orientations and lengths. However, this is very difficult to achieve in practice for character modeling or complex objects. Another approach is to utilize continuous seamless (chartless) parameterization, which is extremely difficult or even impossible to generate.

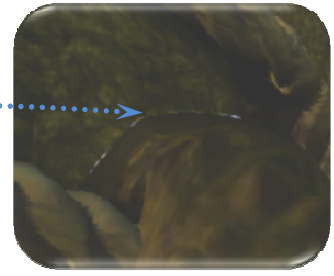
We solve this problem during the map generation process, rather than at run-time, via additional features implemented as part of the GPUMeshMapper tool. We post-process our displacement maps by correcting all the texture uv borders during the displacement map generation, by identifying the border triangle edges and performing filtering across edges (with additional fix-up) to alleviate displacement seams.



Displacement map



Rendered character using the displacement map with a seam



Zoomed-in view shows a crack in the surface

Figure 17. Example of a visible crack generated due to inconsistent values across the edges of displacement map for this character. On the left we highlighted the specific edges along the seam. Note that the adjacent edges for this seam do not have uniform parameterization.

Acknowledgements

We thank the members of AMD's Game Computing application for their work on the "Froblins" demo, specifically Jeremy Shopf, Christopher Oat and Abe Wiley (whose art was used for all images in this article). Additionally, we greatly appreciate the fruitful discussions and suggestions from Nicolas Thibieroz, who also contributed the tessellation tutorial SDK sample on the DVD. Finally, we thank Peter Lohrmann, Budirijanto Purnomo, and Dan Ginsburg for their hard work on the AMD GPUMeshMapper tool.

References

- [AMD09] AMD Radeon SDK and Documents. 2009.
<http://developer.amd.com/gpu/radeon/pages/RadeonSDKSamplesDocuments.aspx>
- [AMDGMM08] AMD GPU MeshMapper Tool. 2008. <http://ati.amd.com/developer/gpumeshmapper.html>
- [BLYTHE06] Blythe, D. 2006. The Direct3D 10 System. *ACM Transactions on Graphics*, 25, 3, pp. 724--734.
- [CASTAÑO08] Castaño, I. Water-Tight, Textured, Displaced Subdivision Surface Tessellation Using Direct3D 11. Presentation. Gamefest 2008, Seattle, Wash., July 2008.
- [CATMULLCLARK78] Catmull, E. and Clark, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Comput. Aid. Des.* 10, 6, 350--355
- [FROBLINS08] AMD ATI Radeon HD 4870 demo, *Froblins*. 2008.
<http://developer.amd.com/documentation/videos/pages/froblins.aspx>
- [GEE08] GEE, K. 2008. Direct3D 11 Tessellation. Presentation. Gamefest 2008, Seattle, Wash., July 2008.
- [KLEIN08] KLEIN, A. 2008. Introduction to the Direct3D 11 Graphics Pipeline. Presentation. Gamefest 2008, Seattle, Wash., July 2008.
- [LOOP87]. Loop, C. T. 1987. Smooth subdivision surfaces based on triangles, 1987. Master's Thesis, Department of Mathematics, University of Utah.
- [LOOPSCHAEFER08] Loop, C. and Schaefer, S. 2008. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. *ACM Transactions on Graphics (TOG)*, V. 27, I. 1.
- [PIEGLTILLER96] Piegl, L.A. and Tiller, W. 1996. *The NURBS Book (Monographs in Visual Communication)*, 2nd Ed. Springer.

- [PERSSON06] Persson, E. 2006. Render to Vertex Buffer Programming. ATI Technologies Technical Report, March 2006. http://developer.amd.com/media/gpu_assets/R2VB_programming.pdf
- [SBCT08] Shopf, J., Barczak, J., Oat, C., and Tatarchuk, N. 2008. March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU. ACM SIGGRAPH 2008: Proceedings of the conference course notes, Advances in Real-Time Rendering in 3D Graphics and Games, [Chapter 3, pp. 52-101, Los Angeles, CA, August 2008. PDF](#)
- [STAM98] Stam, J. 1998. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values, Proceedings of the 25th annual conference on computer graphics and interactive techniques, p.395-404, July 1998.
- [TATARCHUK06] Tatarchuk, N. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (SI3D '06), pp. 63-69. [\[PDF\]](#)
- [TATARCHUK08] TATARCHUK, N. 2008. Advanced Topics in GPU Tessellation: Algorithms and Lessons Learned. Presentation. Gamefest 2008, Seattle, Wash., July 2008.
- [TATARCHUK09] Tatarchuk, N. 2009. Dynamic Terrain Rendering on GPU Using Real-Time Tessellation. ShaderX⁷: Advanced Rendering. Engel, W., Ed. Charles River Media, Cambridge, Mass.
- [VPBM01] Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. 2001. Curved PN triangles, Proceedings of the 2001 symposium on Interactive 3D graphics, pp.159-166, March 2001.
- [ZORINSCHRÖEDER00] Zorin, D. and Schröder, P. 2000. Subdivision for Modeling and Animation. ACM Siggraph 2000 course.