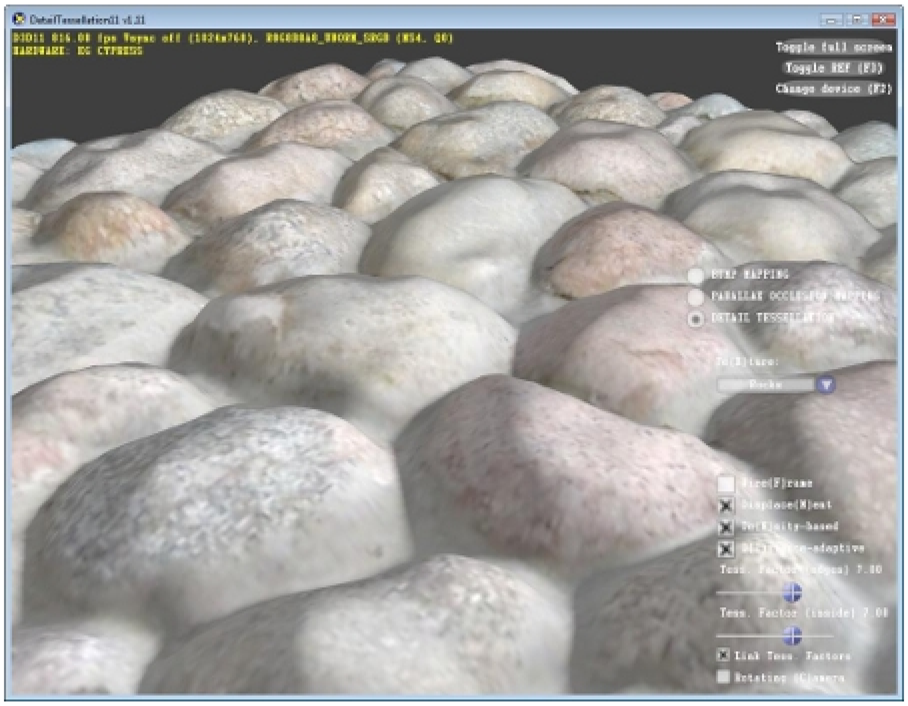
DetailTessellation11 Sample

Collapse All

This sample, contributed by AMD, demonstrates the use of detail tessellation for improving the quality of material surfaces in real-time rendering applications. Detail Tessellation is typically a faster-performance and higher-quality alternative to per-pixel height map-tracing techniques such as Parallax Occlusion Mapping.



```
D3D11 816.08 fps Vsync off (1024x768), R8G8B8A8_UNORM_SRGB (MS4, Q0)
HARDWARE: EG CYPRESS
```

Path

Source	SDK root\Samples\C++\Direct3D11\DetailTessellation11
Executable	SDK root\Samples\C++\Direct3D11\Bin\x86 or x64\DetailTessellation11.exe

DirectX 11 Tessellation

Detail tessellation adds geometric detail onto material surfaces by using DirectX 11 Tessellation to introduce new vertices onto the underlying geometry and displacing those vertices according to an accompanying height map. This approach has multiple advantages compared to traditional "fake" per-pixel displacement techniques such as bump mapping, parallax mapping and even parallax occlusion mapping.

Real displacement

Because the surface's "bumpiness" is modeled using real geometry the quality of the relief produced usually exceeds "fake" per-pixel displacement techniques prone to under-sampling issues. Also, it allows proper interactions with other Z-tested geometry which can be useful for some purposes.

No Aliasing

Unlike per-pixel "ray casting" techniques such as Parallax Occlusion Mapping edges produced from Detail Tessellation are not subject to aliasing issues and benefit from Multisampling Anti-Aliasing.

Accurate depiction of silhouette edges

A significant shortcoming of Parallax Occlusion Mapping is the inability to model silhouette edges from bumps protruding from the surface. Detail Tessellation renders real silhouettes by displacing actual vertices so the geometry produced is not confined to pixels contained within the original triangle.

Performance

Not only does Detail Tessellation produce higher-quality results than traditional per-pixel displacement techniques, it also does so at a significantly inferior performance cost on modern DX11 graphic hardware due to a typical reduction in processing tasks compared to more costly techniques such as Parallax Occlusion Mapping.

Implementation

In its simplest form, detail tessellation is a combination of tessellation combined with displacement mapping. The Vertex Shader calculates position and tangent space light/view vectors before passing them to the Hull Shader along with texture coordinates. The Hull Shader triggers “pass-through” mode by outputting three control points that are a direct copy of the input data (position, texture coordinates and light/view vectors), while the HS Patch Constant function applies a uniform tessellation factor to triangle edges and its inside. The new vertices generated by the fixed-function tessellator are input to the Domain Shader, which calculates their position and displace them using the height map provided. All lighting is performed in the pixel shader using the interpolated tangent space light vector (and view vector if specular is enabled) and normal.

Density-based Tessellation

This sample supports “density-based tessellation”, whereby larger tessellation factors are applied onto geometry mapped to high-frequency regions of the displacement map. Lower-frequency regions have smaller tessellation factors applied to them in order to preserve performance. The computation of the vertex density is an offline process which consists in comparing adjacent values in the displacement map, with large variations resulting in higher densities. All density calculation is eventually performed on a per-edge basis in order to avoid discontinuities when tessellating. This sample uses a simple function for edge density calculation, and better algorithms could be implemented.

When density-based tessellation is enabled the HS Patch Constant function performs the additional task of fetching edge vertex densities from a Buffer resource, and assigns them to the tessellation factors to apply to the geometry for tessellation.

Distance-adaptive Tessellation

Another optimization is to scale the tessellation factors by a distance function. In principle, more details are needed when the camera is close to the geometry. When geometry is further away fewer triangles can be produced without an obvious loss of image quality. This allows performance savings by reducing the tessellation factors to apply as the camera gets away from the geometry. This sample implements distance-based tessellation, and clamps the minimum tessellation factor to a value of half the selected edge tessellation level in order to preserve the minimum level of detail required to accurately represent silhouettes edges from a distance. The distance scaling function should of course be adapted to the needs of each application.

When distance-adaptive tessellation is enabled the Vertex Shader calculates the distance from the current vertex to the camera, and outputs a distance factor into an additional Vertex Shader output (Hull Shader input). The HS Patch Constant function then calculates the distance factor for each edge, and use it to scale the tessellation factors.

Screen-space adaptive Tessellation

In this mode the overall length of each screen-space triangle edge is calculated in the Hull Shader, and the tessellation factor to apply on each edge is a factor of the desired triangle size in pixels. This adaptive scheme is similar to distance-based tessellation, although it requires more instructions in the Hull Shader.

Frustum Culling Optimization

One way to reduce the performance cost associated with the tessellation of multiple primitives is to ensure that tessellation is only applied to input primitives that are actually visible. Therefore a sensible optimization is to do a frustum-culling test of incoming triangles and only tessellate them if they are at least partially visible from the current camera position. This optimization is implemented by extracting the left, right, top and bottom clip plane equations from the view-projection matrix (testing against the near and far clip plane is not performed as they typically contribute little culling) and passing them to the Hull

Shader. The Hull Shader will then test whether a triangle passes each of the plane equations and set tessellation factors to 0 should any of the test fails (a tessellation factor of 0 results in no tessellation). This optimization results in significant performance gains for a moderate additional cost in ALU shader instructions.