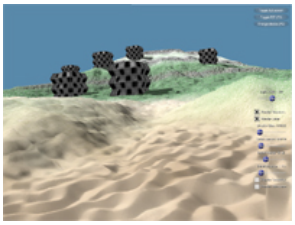
RaycastTerrain Sample

Collapse All

This sample demonstrates a method for rendering terrain that does not rely on underlying geometry. Rays are cast from the eye and intersected directly with the terrain image map using cone-step mapping. In addition, we add extra detail by tiling and blending relief maps over the terrain and doing a relief map step after the initial cone-step intersection.



Path

Source	*SDK root* \Samples\C++\Direct3D10\RaycastTerrain
Executable	*SDK root* \Samples\C++\Direct3D10\Bin\ *x86 or x64* \RaycastTerrain.exe

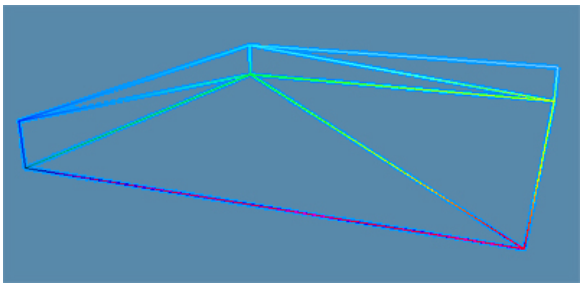
How the Sample Works

This sample uses relief mapping and a variation of relief mapping called cone-step mapping to intersect eye rays directly with image height maps in order to render terrain. The terrain is rendered as a tile where each tile represents the area covered by one image map. While this sample can handle multiple terrain tiles, only one is shown for simplicity.

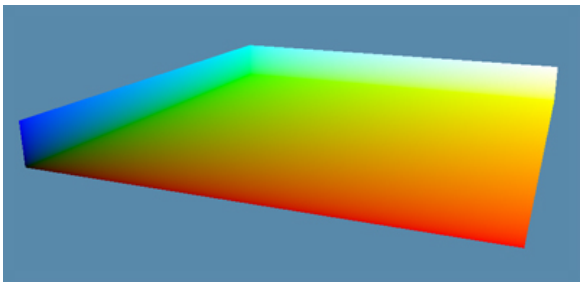
Finding Entry and Exit Points

In order to intersect the eye ray with the terrain, we must first determine where the eye ray enters and exits the terrain tile. To make intersecting the terrain easier, we want these entry and exit points in the texture space of the terrain tile. The entry and exit points are three-dimensional vectors, with x and y representing u and v respectively in texture space. The z coordinate of the entry and exit points represents height above the terrain in texture space with 0 and 1 being the minimum and maximum values.

We use the graphics hardware to find the exit point of the ray by simply rasterizing the inside of a box.



Each corner of the box contains the UV coordinates for the exit point as it would be at that corner. When rasterized, the exit points are interpolated so that each pixel shader invocation receives the correct exit point for the eye ray intersecting that pixel.



While the exit point is easy to compute, the entry point is a little more involved. For the entry point, there are two cases, and we specialize the shader accordingly. The first case is where the eye point is inside the terrain tile volume. In this case the entry point is the eye point transformed into the texture space of the terrain tile. The second case is where the eye point is located outside of the terrain tile volume. In this case, for each pixel rendered we must intersect the eye ray with the terrain tile volume's bounding box. This must be done in the pixel shader. Fortunately, this calculation is simplified if we assume the bounding box is always axis aligned.

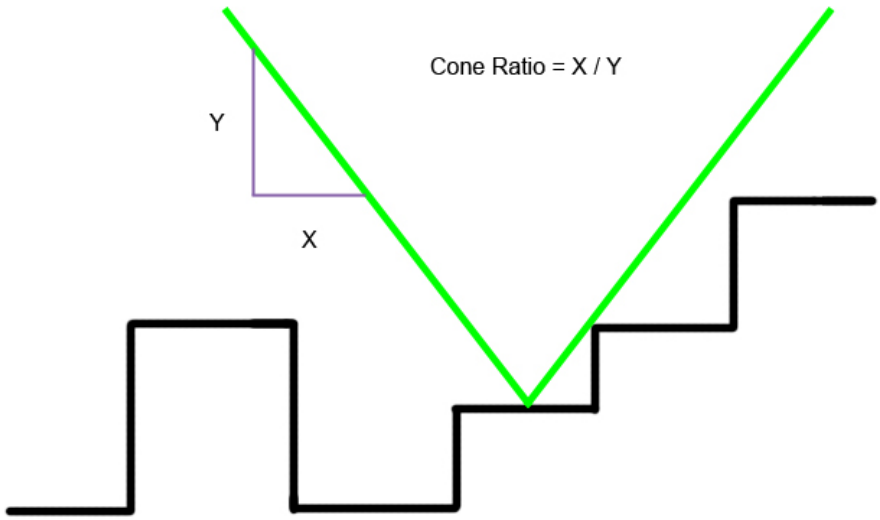
```
//--------------------------------------------------------------------------------------
// Intersect the ray with the texture bounding box so we know where to start.
// This is for the case where our eyepoint is outside of the box.
//--------------------------------------------------------------------------------------
float3 GetFirstSceneIntersection( float3 vRayO, float3 vRayDir )
{
    // Intersect the ray with the bounding box
    // ( y - vRayO.y ) / vRayDir.y = t

    float fMaxT = -1;
    float t;
    float3 vRayIntersection;
```

```
    // -X plane
    if( vRayDir.x > 0 )
    {
        t = ( 0 - vRayO.x ) / vRayDir.x;
        fMaxT = max( t, fMaxT );
    }

    // +X plane
    if( vRayDir.x < 0 )
    {
        t = ( 1 - vRayO.x ) / vRayDir.x;
        fMaxT = max( t, fMaxT );
    }

    // -Y plane
    if( vRayDir.y > 0 )
    {
        t = ( 0 - vRayO.y ) / vRayDir.y;
        fMaxT = max( t, fMaxT );
    }

    // +Y plane
    if( vRayDir.y < 0 )
    {
        t = ( 1 - vRayO.y ) / vRayDir.y;
        fMaxT = max( t, fMaxT );
    }

    // -Z plane
    if( vRayDir.z > 0 )
    {
        t = ( 0 - vRayO.z ) / vRayDir.z;
        fMaxT = max( t, fMaxT );
    }

    // +Z plane
    if( vRayDir.z < 0 )
    {
        t = ( 1 - vRayO.z ) / vRayDir.z;
        fMaxT = max( t, fMaxT );
    }

    vRayIntersection = vRayO + vRayDir * fMaxT;

    return vRayIntersection;
}
```

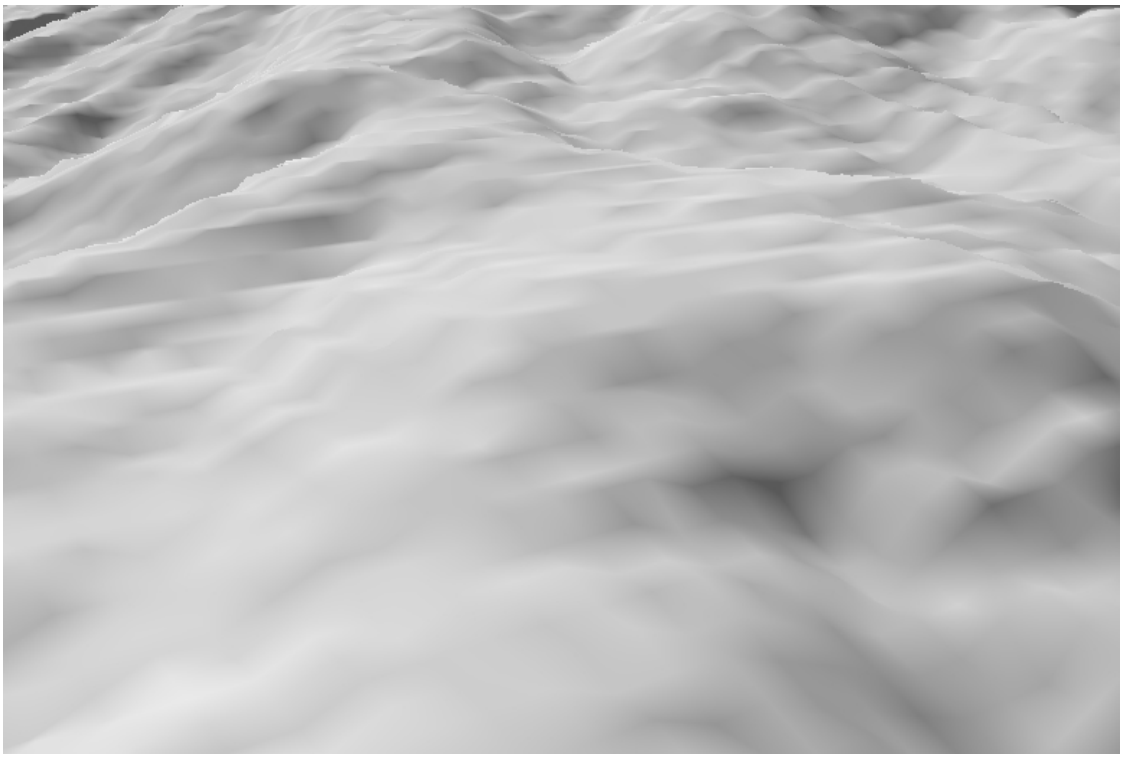
Cone Step Mapping for Coarse Detail

Once we have entry and exit points in texture space, we can raycast through the terrain texture and determine where our ray intersects the height map. Before we explain Cone Step Mapping, let us ponder a simpler method of finding the intersection. One simple approach is to make fixed steps between the entry and exit points and to terminate the search when the z coordinate of our test point is below the value retrieved from the height map. This approach has two main disadvantages. The first is speed, or lack thereof. It involves several unnecessary texture fetches which can become a bottleneck on a large texture. The second is accuracy. If taking fixed steps, the first point at which the z coordinate falls below the height map may not be the exact intersection.

To solve the first issue, we use a variation of relief mapping called cone-step mapping to skip large areas of texture space in search of the intersection. Cone-step mapping involves a preprocessor phase. An imaginary inverted cone is placed on each height map texel and the cone radius is expanded until another part of the height-field is hit.

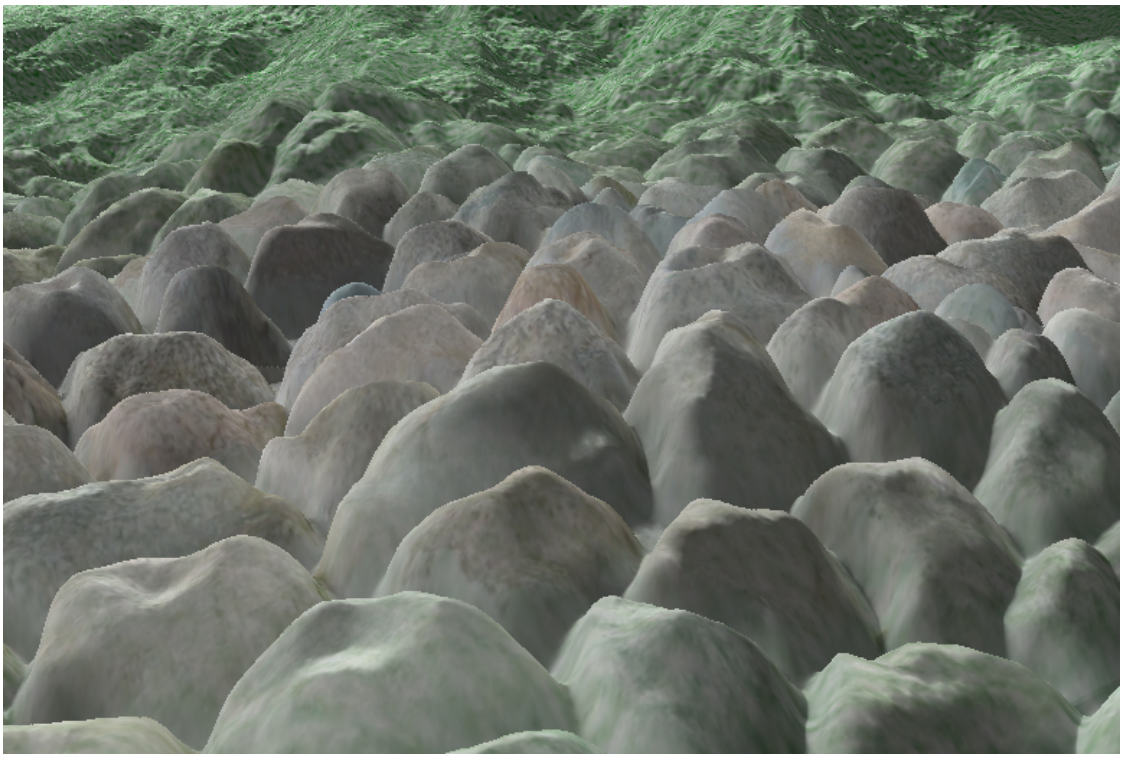


Anything inside this cone can safely be excluded from the search since it is guaranteed that no other part of the height field intersects this cone. When tracing, we use the intersection of the eye ray with each cone we've sampled to determine the next location from which to continue our trace. Cone-step mapping gives us an approximate intersection point and a coarse representation of the underlying geometry. If no intersection point is found before we reach the exit point, we discard the pixel. This gives plausible silhouettes for the terrain.

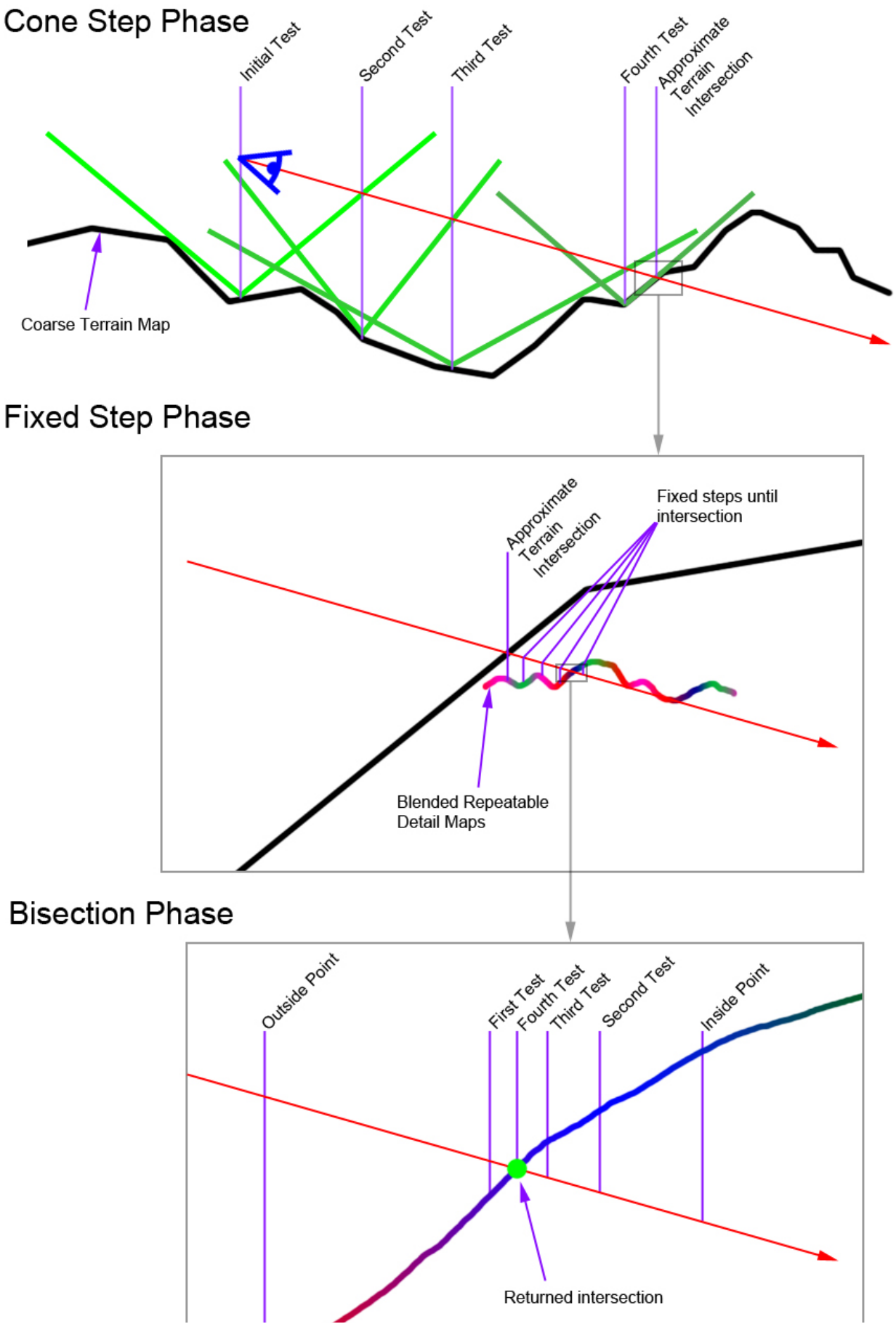


Adding Fine Detail With Relief Mapping

Cone-step mapping gives a good approximation of the overall terrain shape. However, for finer detail, we tile several smaller height map textures over the terrain and blend them together using 4 component mask texture. We then trace through these textures starting at the intersection point previously given by cone-step mapping. Unfortunately, we cannot use cone-step mapping to accelerate this phase. This is because several individual height maps are blended together in an arbitrary fashion, and our preprocessing step that was required for cone-step mapping cannot handle the unpredictable nature of the blend. For this phase, we use standard relief mapping. We start at the intersection point given to us by cone-step mapping and then trace into the tiled and blended detail height map textures. We start with fixed sized steps to find the first rough intersection. We then use a bisection phase (binary search) to find the exact location of intersection. We've essentially added relief mapped details to an already relief mapped terrain.



The following diagram gives an overview of the entire raycasting process.


Cone Step Phase
Initial Test
Second Test
Third Test
Fourth Test
Approximate Terrain Intersection
Coarse Terrain Map
Fixed Step Phase
Approximate Terrain Intersection
Fixed steps until intersection
Blended Repeatable Detail Maps
Bisection Phase
Outside Point
First Test
Fourth Test
Third Test
Second Test
Inside Point
Returned intersection

Interaction with Triangulated Geometry

Because we're doing all of the intersections in texture space, we need a way to make sure that the raycast terrain can still interact with polygonal geometry. We handle two types of interaction: Depth intersection, and shadows. In order for the terrain to occlude polygonal geometry and vice versa, we need to output accurate depth values from the pixel shader. In order to do this, we need to move the final intersection point from terrain tile texture space to clip space. The output z coordinate of the transformed point divided by the homogenous w coordinate is the depth that we write directly into the depth buffer.

```
// Convert the intersection point to a depth value
float4 vProjPoint = mul( float4( vIntersection.xzy, 1 ), g_mTexToViewProj );
Output.depth = vProjPoint.z / vProjPoint.w;
```

Shadows

For shadows, we could simply cast a ray from the intersection point toward the light and determine if we hit any more terrain before hitting the light source. While this would give accurate terrain self-shadowing, we would be missing some important shadow cues, namely Shadows from the terrain onto polygonal objects and from polygonal objects onto the raycast terrain. Because we can output accurate depths from the raycast terrain, we can use traditional shadowmapping to solve the problem. We simply render the terrain and polygonal objects from the point of view of the light, and because we have accurate depths, we can use these as a shadow depth map when rendering the objects from the camera point of view.

References

[1] Dummer, Jonathan. "Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm." http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[2] Policarpo, Fabio, Manuel M.Oliveira. "Relaxed Cone Stepping for Relief Mapping." *GPU Gems 3*, Addison-Wesley, Upper Saddle River, NJ. p. 409-427. 2007.

[3] Oliveira, Manuel, M. Gary Bishop, David McAllister. "Relief Texture Mapping." *Proceedings of SIGGraph 2000*, p. 359-368, August 2000.