White Paper

Clipmaps

February 2007
WP-03017-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
	February 15, 2007	EM, TS	Initial release

Go to sdkfeedback@nvidia.com to provide feedback on Clipmap.

Clipmaps

Abstract

Clipmaps are a feature first implemented on SGI workstations that allow mapping extremely high resolution textures to terrains. The original SGI implementation required highly specialized, custom hardware. The advanced features of the NVIDIA® GeForce® 8800 now permit the same algorithm using consumer hardware.

Although current APIs and the GeForce 8800 directly support textures with dimensions up to 8192, this size may be considered insufficient when we talk about wide landscapes, say, in flight simulators. The idea of using a single texture for the whole landscape can be very promising due to the fact that we can not only design the whole landscape texture at once, but also parameterize it simply. Big textures have "big" advantages compared to traditional methods of using several textures with blending. This comes from the fact that they can be as complex as you wish. Ones a designer has created a whole map it can be used as is.

Clipmaps take advantage of the fact that, due to perspective projection, only relatively small regions within the texture mipmap pyramid are being accessed every frame. Thus we have to manage these "hot" regions and update them in video memory as the viewer moves around. A DX10 solution is to store such regions in a texture array. Being able to index into it from the pixel shader allows for a straightforward implementation of the clipmap algorithm in DX10.

Evgeny Makarov
NVIDIA Corporation

How Clipmap Works

Clipmap can be defined as a *partial representation of a mipmap pyramid which holds all information needed for texturing at every single frame*. How do you determine which data from a source texture can potentially be used? The answer lies in a mipmap sample selection strategy. The best case while texturing is one that allows you to use 1:1 mappings of texels to pixel area. That is how you can define clip size for mipmap levels based on the current screen resolution. The lowest levels of the mipmap pyramid will always fit in video memory and can be used statically. All other mip levels form the clipmap stack which is dynamically updated to store actual data at every frame (see Figure 1). The contents of a stack in most common cases can be defined by its size and the viewer's position.

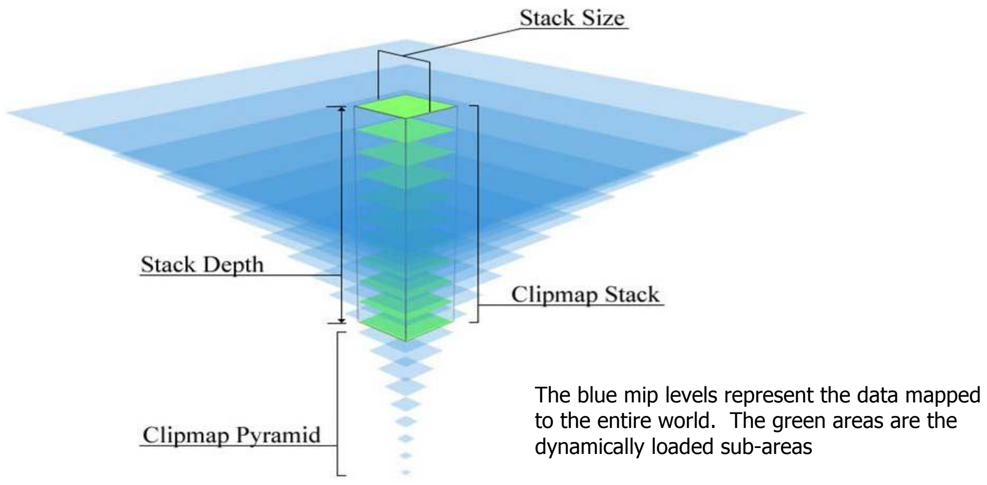
Stack Size
Stack Depth
Clipmap Stack
Clipmap Pyramid
The blue mip levels represent the data mapped to the entire world. The green areas are the dynamically loaded sub-areas

Figure 1. Clipmap Representation

The basic idea is to store the clipmap stack in a 2D texture array. Texture arrays are a new feature of DX10. The remaining part of the mipmap pyramid is implemented as a conventional 2D texture with mips. You can perform a dynamic stack update using copy/update sub resource methods. It is totally clear that sometimes it would not be possible to hold all the data needed in system memory. Therefore you are going to need an additional mechanism to stream all necessary data efficiently from disk.

Data Representation

A clipmap stack is stored in a 2D texture array. This array forms a dynamic part of clipmap and should contain actual data for every mip level for each frame. Since there are separate layers for each original mip level, you should create a texture without mips. The remaining part of the image can be stored as a conventional 2D texture.

Using the DX10 API, create these resources as follows (note that for a clipmap stack texture, you should specify the number of layers using the **`ArraySize`** element):

```
D3D10_TEXTURE2D_DESC texDesc;

ZeroMemory( &texDesc, sizeof(texDesc) );
texDesc.ArraySize = 1;
texDesc.Usage = D3D10_USAGE_DEFAULT;
texDesc.BindFlags = D3D10_BIND_SHADER_RESOURCE;
texDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texDesc.Width = g_PyramidTextureWidth;
texDesc.Height = g_PyramidTextureHeight;
texDesc.MipLevels = g_SourceImageMipsNum - g_StackDepth;
texDesc.SampleDesc.Count = 1;

pd3dDevice->CreateTexture2D(&texDesc, NULL, &g_pPyramidTexture);

texDesc.ArraySize = g_StackDepth;
texDesc.Width = g_ClipmapStackSize;
texDesc.Height = g_ClipmapStackSize;
texDesc.MipLevels = 1;

pd3dDevice->CreateTexture2D(&texDesc, NULL, &g_pStackTexture);
```

Updating Strategy

As you move around, you need to update the content of the stack based on a new clip center position. In most cases you should replace relatively small portions of data in each layer of a clipmap stack. To avoid big data replacement within a stack, use a special technique known as *toroidal addressing*: Toroidal addressing is new data at the top of the image is loaded at the bottom, and data on the right is loaded at the left. This approach does not need any changes for overlapped regions, which is a big plus in this case.

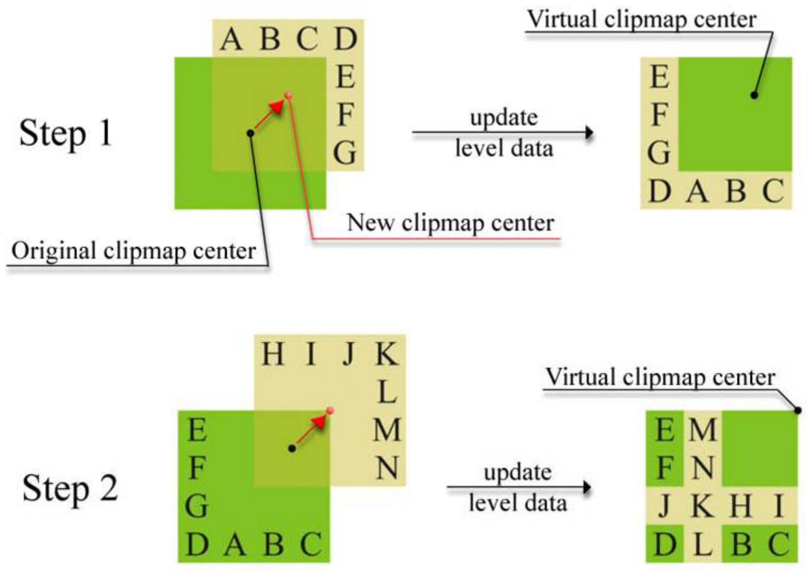

Figure 2. Two Update Steps for a Single Layer in a Stack

In most applications, this process can be even simpler because you can separately update the horizontal and vertical part resulting in simple rectangular regions instead of L-shaped ones.

Clipmap Texture Addressing

All the work is done in the pixel shader. First you need to determine a mip level to fetch from. For this use the **ddx** and **ddy** instructions to find the quad size in a screen space.

```
float2 dx = ddx(input.texCoord * textureSize.x);
float2 dy = ddy(input.texCoord * textureSize.y);

float d = max( sqrt( dot( dx.x, dx.x ) + dot( dx.y, dx.y ) ) ,
  sqrt( dot( dy.x, dy.x ) + dot( dy.y, dy.y ) ) );
```

Now you can easily calculate a suitable mip level as follows.

```
float mipLevel = log2( d );
```

Calculate the **mipLevel** as a float and use the fractional part to perform trilinear filtering.

Clipmap texture addressing is rather simple; the only thing you need to do is to scale the input texture coordinates based on the mip level. Calculate a scale factor by dividing the source image size by the clipmap stack size.

```
float2 clipTexCoord = (input.texCoord) / pow(2, iMipLevel);
clipTexCoord.x *= scaleFactor.x + 0.5f;
clipTexCoord.y *= scaleFactor.y + 0.5f;
float4 color = StackTexture.Sample( stackSampler,
  float3(clipTexCoord, iMipLevel) );
```

For the stack sampler, specify the address mode as *wrap* to implement toroidal addressing.

Table 1. Storage Efficiency*

Texture sizes	**4096^2**	**8192^2**	**16384^2**
Full mipmap	85.3	341.3	5461.3
1024 clipmap	13.3(16%)	17.3(5%)	25.3(<1%)
2048 clipmap	37.3(44%)	53.3(16%)	85.3(1.6%)
4096 clipmap	85.3(100%)	149.3(44%)	213.3(3.9%)

*Memory costs for 32-bit texels storage

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com