Tiled Resources 11

Matt Lee, Senior Software Development Engineer

The Tiled Resources 11 example implementation demonstrates *tiled resources*, which are resources whose residency can be controlled on a finer granularity than traditional graphics resources. Similar technologies implemented by others have been called *megatextures*, *virtual textures*, *sparse textures*, and *partially resident textures*. This particular implementation is specifically designed to validate a proposed hardware implementation of the tiled resources feature.

# Introduction

Tiled resources are a proposed solution to a content problem, which is that large textures are impractical to use in today's graphics engines and hardware. Today, textures and other graphics resources must be fully resident to be used, which is a problem when a large texture spans tens or hundreds of megabytes of memory—or even more. Besides the fact that streaming such a large amount of data is impractical due to loading times, once such a texture is loaded, it is largely wasted, frame to frame, since there isn't a way to utilize a significant fraction of such a large texture for a single frame. For example, a 16384×16384 texture has 268 million texels in its base mipmap level, but a 1080p frame only has a bit over 2 million pixels.

Software solutions, such as megatextures, virtual textures, and other proprietary software systems, solve this problem by splitting large resources into small pieces and using an indirect fetching mechanism to simulate a sparsely populated large texture. These solutions effectively solve the problem of large resource content, but they incur performance penalties and/or hit restrictions on resource usage that may not be acceptable for broad usage.

This implementation proposes a theoretical hardware implementation that would define key attributes and behaviors of tiled resources, with an eye towards taking maximum advantage of GPU hardware and software features that already exist in modern GPUs, such as memory management units and page tables. This software implementation was designed to validate this proposed GPU hardware implementation. As a result, this implementation does not represent an optimal software implementation of this feature.

# Using the Implementation

There are two views in the application: a *swatches* view that shows several texture format swatches, and a *terrain* view that shows a large terrain object textured with a tiled texture. The “Terrain render” checkbox in the lower right corner of the window switches between the two views.

The terrain view depends on three large texture content files that are generated by Tiled Texture Pack Tool, an example content tool that is distributed with Tiled Resources 11. If the texture files are not present, generate the texture files by using the tool, and then follow the on-screen instructions in the application to deploy the texture files to the right place on your PC.

In the swatches view, dragging with the left mouse button will move the camera. The mouse scroll wheel and/or the W and S keys are used to zoom in and out. In the terrain view, dragging with the left mouse button changes the yaw and pitch of the camera, and the W, A, S, and D keys are used to move the camera forwards, backwards, and strafe left and right. Holding down the left Shift key will reduce the camera movement speed.

Information about the streaming system is displayed in the upper left corner of the screen, and in the swatches view, information about the active swatch in the middle of the screen is detailed in the top left of the screen when a swatch is right clicked.

The residency and streaming systems can show debugging information onscreen to further demonstrate streaming behavior. When a swatch is selected with a right click, the tile-streaming view for the active swatch is shown. If the tile-streaming view scrolls off screen, the PageUp and PageDn keys can be used to scroll the tile-streaming view. The current residency view can be displayed by toggling the “Show residency view render” checkbox in the lower right. In terrain view, information about the terrain’s diffuse map texture can be displayed by right clicking anywhere in the view. Finally, the “Pause streaming” checkbox pauses and resumes streaming activity, except for any tile loading in progress.

# Proposed Tiled Resource Hardware Notes

The proposed hardware implementation would have the following features:

* All tiles in tiled resources are 64KB in size. The layout and size of each tile depends on the format and resource dimension of the tiled resource.
* The tiled resource software API implements a tile pool, which is a collection of 64KB tiles. Tiles can be mapped to tiled resources by specifying the tile index within the tile pool and a handle that specifies the location within the tiled resource.
* Tiled 2D textures are split into tiles using a simple rectangular pattern, starting at the upper left hand corner of each mipmap level.
* When the hardware samples from a region of a tiled resource that is not resident, the resulting value is zero (instead of a hardware failure), and a residency status flag is set that can be queried by HLSL shaders. The hardware does not implement any automatic fallback mechanisms in the event of a nonresident fetch; it is entirely up to the shader author to handle nonresident fetches in their own implementation-specific way.
* The hardware is capable of fetching across tile boundaries in the case of bilinear, trilinear, or anisotropic filtering. Again, if one or more of the samples fetched are nonresident, zero results will be used for those samples, and a residency status flag will be set accordingly.

# Example Implementation Architecture Notes

The example implementation can be thought of as having three logical sections, described in the following table.

|  |  |
| --- | --- |
| Section | CPP/H/HLSL files |
| Software and APIs  The software and APIs that create and manipulate tiled resources, tile memory, and the "tile table entries" that map tiled resources to tiled memory. | d3d11tiled  pagerenderer  physicalpagemanager  tiledresourcebase  tiledresourcecommon  tiledresourced3d11  tiledresourceruntimetest  typedpagepool |
| Emulation by the shader software  The shader software's emulation of proposed hardware tiled texture sampling hardware. | tiledresourceemulationlib.hlsl |
| Title systems  The game title systems that are required to exercise the tiled resources feature and other features that are endemic to the example implementation itself. | tiledresource11  pageloaders  pagedebugrender  residencysamplerender  samplingqualitymanager  sceneobject  terrainview  threadsafequeue  util  tiledresources.hlsl  scenerender.hlsl  DXUT sample framework common code |

The first section is wrapped by the tiled resource API in d3d11tiled.h. The tiled resource API is styled to be consistent with Direct3D 11. The inner workings of the software implementation are contained within the files in the first section, software and APIs, but the code in the third section, title systems, interacts with the software implementation exclusively through the d3d11tiled API wrapper.

Tiled resources are implemented as a pair of textures: an *index map texture*, which represents virtual-to-physical tile memory mappings, and a *physical page array texture*, which holds the physical tile memory contents as atlased sections of an array texture. There is one index map texture per tiled resource, and one physical page array texture for each tiled resource data format. Therefore, two tiled textures of the same format would have individual index maps and share the same physical page array texture. The tiled resource API abstracts the index map and physical page array textures, and presents them as an untyped tile pool and individual mappings from virtual tile locations to physical tile indices.

# Performance Notes

This example implementation is intended as a software implementation of a proposed hardware architecture for tiled resources, and as designed, it is not an optimal software implementation. Several steps of the algorithm are performed at run time that could either be performed offline or eliminated entirely in order to accelerate performance. If you wish to use this example implementation as the starting point for an optimized software implementation, consider the following improvements:

* Remove the 1:1 correspondence between tiled texture sampling, index map sampling, and physical page array texture sampling. You may only need one index map texture and several physical page array textures for different surface shading channels (diffuse, normal, specular, and so on). A similar reduction can be done with the sampling quality textures.
* This sample manages border texels at run time to reduce content complexity, and reflect the idea that the proposed tiled resource hardware would not require any data replication in order to properly sample across tile boundaries. For maximum performance, the border texels could be produced offline.
* Trilinear texture samples are emulated using two index map samples and two bilinear physical page array texture samples. A more sophisticated arrangement for the physical page array texture could be devised to natively support trilinear sampling with only one index map texture sample and one trilinear physical page array texture sample.
* Some code and data complexity is required to handle tiled texture sizes that are not powers of two. Non-power-of-two tiled textures have a more complex tile layout in the mipmap chain and often introduce subtle rounding errors in the pixel shader math; this complexity can be avoided if non-power-of-two tiled textures are not needed.

# Game Title Systems

Tiled Resources 11 implements several streaming systems that should look similar to traditional texture-streaming systems. In this case, instead of streaming entire textures in and out of memory, the streaming system deals with smaller 64KB tiles.

The core of the streaming system is in the title residency manager, which maintains a list of loaded tiles, priority queues for loading and unloading, and also parses incoming residency data that informs the priority ordering for streaming. The residency sample renderer draws a low-resolution, simplified view of the scene, which generates per-pixel residency data that indicates which *tiles* of which mipmap levels of which *textures* are visible from the current camera angle. The sampling quality manager maintains a lookup texture that tracks mipmap level of detail (LOD) residency per region across an individual tiled resource; this lookup texture is then used to smoothly lerp in newly loaded tiles in the final scene render.

# Example Implementation Execution Overview

There are several concurrent, multithreaded, time-dependent systems running in the example implementation. This section describes the high-level data flow in each system.

## Tile Residency Loop

Each frame, the following steps are taken to determine which tiles of which resources are visible, and feed that data into the title residency manager for tracking:

1. Render the scene at a low resolution (256×144), and produce the following data sample for each pixel:
   * Resource set ID at the pixel (resource set is one or more textures that are sampled with the same UV space).
   * UV and slice coordinates at the pixel.
   * Minimum texture gradient at that pixel, allowing for a LOD computation later.
   * Expanded UV coordinates at the pixel (whole number component of U and V), for quilted content.
2. Select a pseudorandom subset of the data samples from the previous frame, and compute the following:
   * Resource list for the data sample (list of tiled texture objects).
   * Mipmap LOD at the pixel for each resource.
3. For each combination of <resource, mipmap LOD, UV coordinates, slice index>, do the following:
   * Determine the virtual address of the tile for the given UV coordinates, the slice index, the mipmap LOD, and the resource.
   * Find or create a tile-tracking structure for this data in the title residency manager.
   * On the tracking structure, increment the sample count by one, and set the timestamp to the current time.
   * Increase the mipmap LOD by one, and repeat these steps until you hit the max LOD for the resource.

## Tile Tracking Loop, Loaders, and Unloaders

The following operations are done each frame by the title residency manager. The title residency manager coordinates tile-streaming operations between a sorted *tile tracking list* that tracks all active virtual tiles, a tile loading priority queue that feeds the tile-loader worker threads, and a tile unloading queue that feeds the tile unloader worker thread.

1. Sort the tracked tiles by their *priority score*, which is computed from the sample count, time since last seen, LOD value, and observed proximity to center of screen.
2. For each tracked tile, look at its state, and see if it should be promoted to the next state:
   * *Seen* tiles are queued for loading at their current priority score if their sample count exceeds a threshold.
   * *Queued for loading* tiles are re-queued for loading if their priority score has changed dramatically since the time when they were first queued for loading.
   * *Loaded and mapped* tiles are queued for unmapping if they haven't been seen for a while
   * *Unmapped* tiles are removed from tracking entirely.

Workers on loader thread promote tiles from *queued for loading* to *loading* and then *loaded and mapped*. The loader threads call into the tile pool to create physical tiles, fill physical tiles with texel data, and then map the physical tiles to virtual addresses. The unloader thread simply unmaps the virtual address and sends the physical tile into a tile recycle queue for reuse in a new location.

## Tile Loaders

There are three tile-loader types: Mandelbrot, Color, and File. The first two generate procedural texture data, either pieces of the Mandelbrot set, or solid/checkerboarded colors. Texture compression is done at run time for DXT formats, using the XGraphics surface compression functions. The file loader loads from specially formatted tiled texture files, generated by the Tiled Texture Pack tool.

## Sampling Quality Loop

For each resource, a separate texture is maintained called the *sampling quality texture*. This non-mipmapped, single-channel texture is sized for the tile dimensions on the base level of the parent texture. For example, a 16384 × 16384 32 bpp tiled texture2D with mipmaps has a tile layout of 128 × 128 tiles on the base level, so the sampling quality texture is a R16 texture with 128 × 128 dimensions. If the parent resource is an array texture, so is the sampling quality texture.

Each texel in the sampling quality texture is a 16-bit float value, ranging from 0 to the maximum mipmap level of the parent texture, and it represents the lowest loaded mipmap level in that area of the parent texture. Since it is a floating point value, it can be smoothly animated as new tiles are loaded in the parent resource.

Each frame, each texel in the sampling quality texture is updated by executing the sampling quality pixel shader. The initial value of each pixel is the maximum LOD value for the parent resource (in the case of the 16384 × 16384 texture, the max value is 8.0). The pixel shader executes the following steps:

1. Determine the current fractional LOD value by sampling the texture.
2. Create a test LOD value by subtracting an amount from the current LOD value. The subtracted amount is proportional to the elapsed frame time multiplied by a speed value.
3. Attempt to sample the parent texture using the test LOD as a fixed LOD value.
4. Look at the boolean result of the **GetResidencyStatus** intrinsic function. If the result is true, then the sample succeeded; write out the test LOD value to the render target and terminate.
5. If the residency status result is false, then the test sample failed. Round up the test LOD value to the nearest whole number and try again.
6. If the test fails again, add 1 to the test value, sample from the texture, and repeat until the sample succeeds (**GetResidencyStatus** returns true).
7. Write the final test value to the render target and terminate.

This loop means that the sampling quality texture is always trying to smoothly push each texel value downwards until it hits 0, the base level. When new tiles are mapped in the parent texture, the sampling quality texture's values in those regions smoothly lerp to the new minimum LOD at the given location in the parent texture. When tiles are unmapped in the parent texture, the value immediately ratchets higher by whole numbers to reflect the higher minimum LOD value in those areas.

## Scene Rendering

For each tiled texture, we first sample the corresponding sampling quality map to get our minimum LOD clamping value. Note that we can't sample the sampling quality map with a standard bilinear filter, as that could generate invalid values along texel boundaries. Instead, we use point filtering, or alternatively, a maximum reduction filter that returns the maximum value of the four samples in a bilinear sample pattern.

Once the minimum LOD clamping value is computed from the texture, we can then feed that into a MinLOD variant of a tiled texture sampling function, such as TiledTex2D\_Trilinear\_MinLOD. That returns a color value, which is used as input to the surface shading algorithm to produce the final output color value.

The tiled texture sampling functions in tiledresourceemulationlib.hlsl implement various combinations of sampling features, but all functions share common functionality for sampling from a tiled resource. The common functionality includes the following steps:

1. The mipmap LOD is computed against the tiled texture, by computing the LOD against the index map texture and biasing by the amount required to treat the index map texture as the larger tiled texture. (For FixedLOD function variants, this value is passed in, not computed.)
2. The incoming UV coordinates are adjusted and then used to sample the index map texture. The resulting index map sample represents a location within the physical page array texture.
3. The incoming UV coordinates are manipulated several times to convert from resource UV space to tile UV space, based on the dimensions of the tiled resource. The coordinates are then adjusted to include a border offset, and then further transformed to match the atlas location within a slice of the physical page array texture.
4. The physical page array texture is sampled using the transformed UV coordinates and an array slice index derived from the index map sample.

Note that trilinear sampling involves two passes through the preceding steps, with two adjacent whole number LOD values computed in step 1. The final sample is blended from the results of the two passes using a lerp value derived from the difference between the computed fractional LOD value and the two adjacent whole number LOD values.

# Licensing

The Tiled Resources 11 example implementation is subject to the same EULA as the DirectX SDK (June 2010) release, except for the DirectXTex library (located in TiledResources11\DirectXTex), which is bound to the [Microsoft Public License (Ms-PL)](http://www.microsoft.com/en-us/openness/resources/licenses.aspx#MPL).

© 2011 Microsoft Corporation. All rights reserved.  
Send feedback to [gameds@microsoft.com](mailto:gameds@microsoft.com?subject=Documentation%20Feedback:%20Tiled%20Resources%20Sample).