Sample Code

Streaming large images with Metal sparse textures

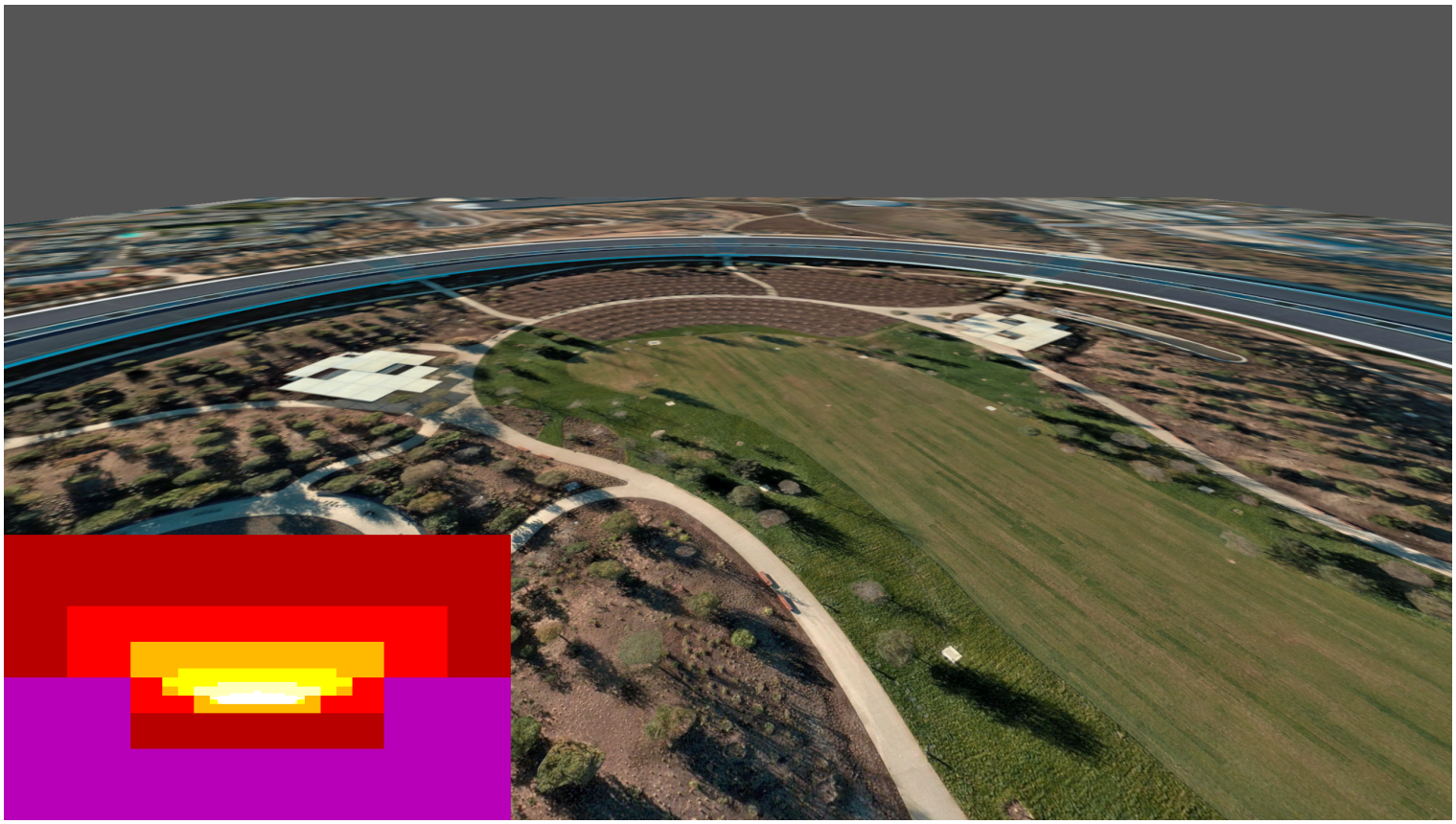
Limit texture memory usage for large textures by loading or unloading image detail on the basis of MIP and tile region.

Download

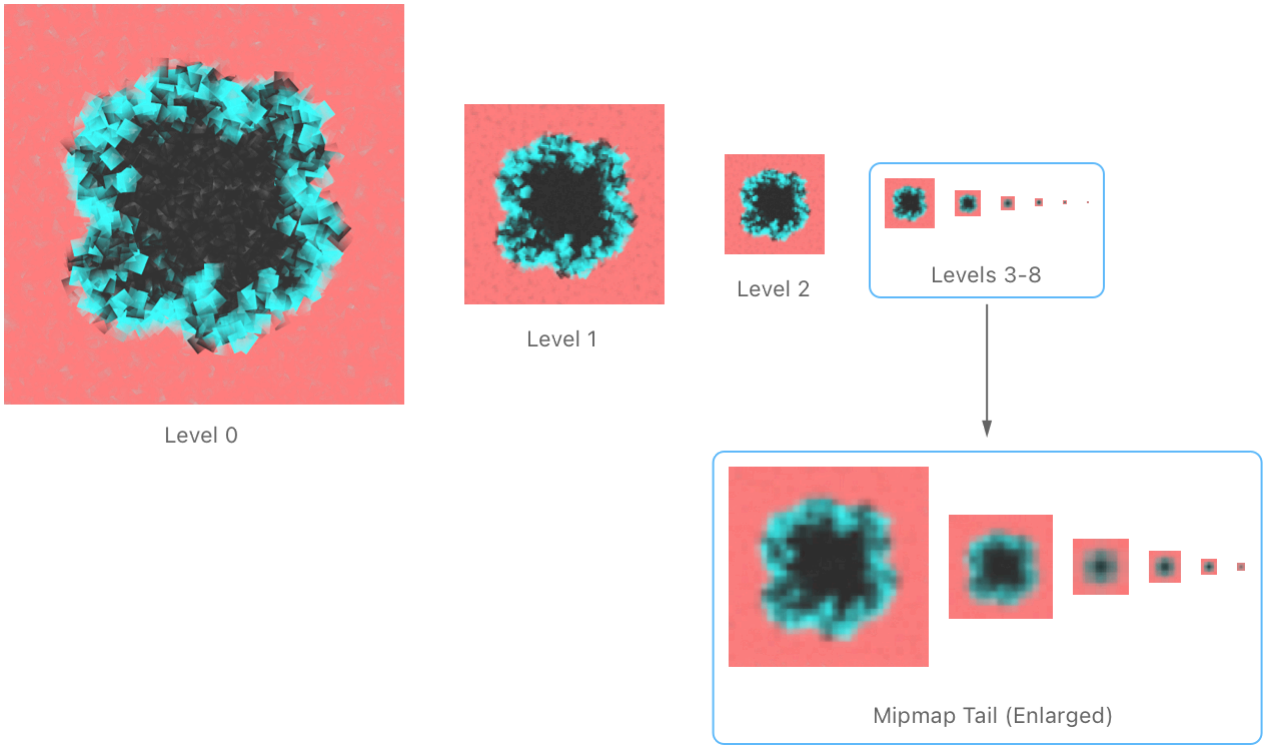
iOS 14.0+ | iPadOS 14.0+ | macOS 11.0+ | Xcode 14.0+

Overview

This sample demonstrates sparse texture streaming by rendering a ground plane that samples from a 16K resolution texture. The renderer uses Metal sparse textures to subdivide the image into regions, or *tiles*, and chooses the tiles to keep in memory. The GPU updates an access counter buffer, and the app determines the tiles it needs to load or discard. The sample shows a heat map of the available MIP levels on the lower left of the screen, where *white* represents level 0, *yellow* represents levels 1 to 3, *red* represents levels 4 and 5, and *purple* represents the remaining MIP levels. The app contains a checkbox that toggles the camera animation. When the animation runs, the app updates the sparse texture as the camera moves through the scene. Lastly, this sample demonstrates asynchronous updates using Dispatch, or *Grand Central Dispatch (GCD)*, to update the sparse texture.



Sparse textures are special textures that manage the residency of both tiles and MIP levels. For instance, a 16K resolution texture may use more than one gigabyte of memory, not including mipmaps that may increase levels memory requirements by 33%. To efficiently use space, the smallest MIP levels are often stored together, called a mipmap tail. For example, this may contain the 8 x 8, 4 x 4, 2 x 2, and 1 x 1 MIP levels. The following figure shows an example texture with its mipmaps and mipmap tail.



The app follows a straightforward process to manage a sparse texture. First, it checks for sparse texture support. Next, it initializes the sparse texture by loading a texture of Apple Park, and

loading and mapping the mipmap tails. Then, the app renders a scene that uses the sparse texture. After rendering, the app updates the texture in parallel with the main render pass. It retrieves the access counters, processes them, and discards tiles that aren't needed anymore. It also maps and unmaps tiles, blits nonresident tiles, and updates the residency buffer when the blitting work finishes. To *blit* means to copy a rectangle of pixels from a source image buffer to a destination memory buffer.

Configure the sample code project

The Xcode project contains schemes for running the sample on macOS and iOS with a physical device that supports sparse textures. You can enable or disable camera movement by checking the switch button on the top-right of the app screen.

To run the app:

- Build the project with Xcode 12 and later.
- Target a macOS device with an M1 chip or later and macOS 12 or later.
- Target an iOS device with an A13 chip or later and iOS 14 or later.

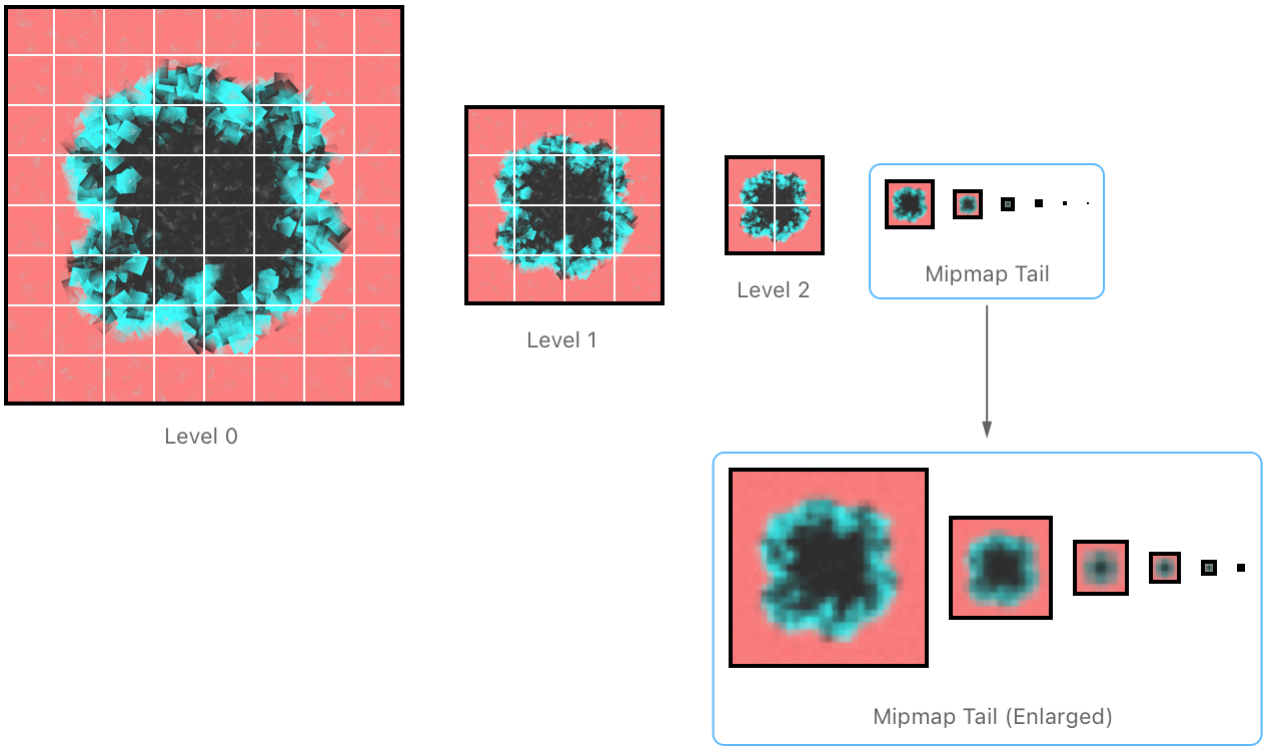
Check for sparse texture support

The sample checks if the `MTLGPUFamily.apple6` feature set is available with the `supportsFamily(_:)` method. This feature set begins with the Apple A13 GPUs. Here's the code from `AAPLViewController:viewDidLoad`:

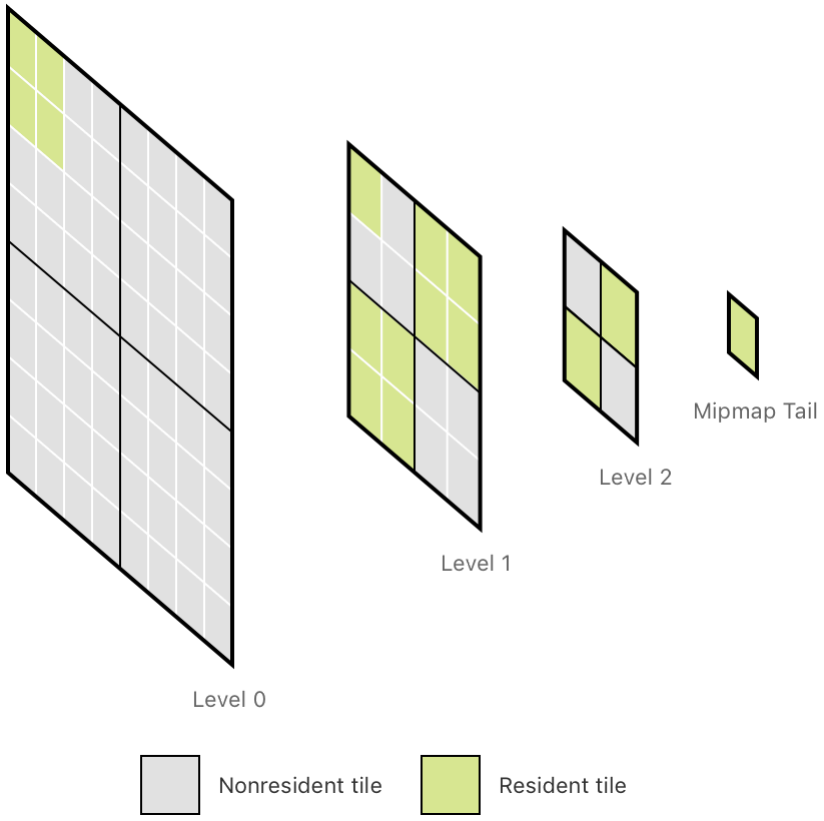
```
// Metal sparse textures require at least MTLGPUFamilyApple6.
if (![_mtkView.device supportsFamily:MTLGPUFamilyApple6])
{
    NSAssert (0, @"This device doesn't support Metal sparse textures.");
}
```

Manage the sparse texture

A sparse texture divides large textures into tiles that the application treats as smaller textures with their own MIP levels. The sparse texture contains a residency buffer that tracks the MIP levels that are currently loaded. The following figure shows how the above texture would be subdivided into separate tile regions. The mipmap tail is considered its own tile, and the app ensures that all the tails are resident.



The residency buffer and access counter buffers use the same layout, but use different data types. The layout is an array of integer values representing each tile, starting with level 0. The tiles are laid out left to right and top to bottom. The app updates the residency buffer while the GPU updates the access counters. The residency buffer uses 8-bit integer values to represent Boolean residency or nonresidency, and the access counters are 64-bit integer values. The following figure shows the memory layout of the residency and access counter buffers.



The `AAPLSparseTexture` class manages the sparse texture in this sample and uses an `MTLHeap` to store the texture data for the tiles. A heap is a Metal object that allows an app to quickly allocate and free textures from a memory pool. Heaps allow quick allocation of tile memory

and help limit the amount of memory used by the sparse texture. In addition to the heap, the class allocates two buffers. The first buffer is the residency buffer that tracks the highest MIP-level resident in the texture. When a shader fails to sample a sparse texture, it can use this buffer to fall back to a resident tile at a lower MIP level.

During rendering, the GPU uses the access counters buffer to store a counter per tile and increments it when a shader samples from the corresponding tile region. The app can query and analyze this buffer to find tiles to map or unmap. When the heap is low on available memory, the class can replace resident tiles that the shader hasn't recently accessed. And this is how the app uses the residency and access counters buffers to dynamically adjust the residency of the sparse texture while staying within a memory budget.

Initialize the sparse texture

The app uses a 16K texture map of Apple Park stored in the Khronos Texture (KTX) file format. The `AAPLSparseTexture` and `AAPLStreamedTextureDataBacking` classes manage all aspects of using sparse textures. The app specifies a heap size of 16 MiB to quickly allocate memory to store tile data.

```
#if USE_SMALL_SPARSE_TEXTURE_HEAP
const NSUInteger heapSize = 2 * 1048576;
#else
const NSUInteger heapSize = 16 * 1048576;
#endif

NSURL* _sparseTexturePath = [[NSBundle mainBundle] URLForResource:@"apple_park.ktx"
_sparseTexture = [[AAPLSparseTexture alloc] initWithDevice:_device
                                                      path:_sparseTexturePath
                                              commandQueue:_commandQueue
                                                  heapSize:heapSize];
```

In the following code, the app starts loading the KTX file. The loader reads the file header and maps the file to memory using `mmap`. Memory mapping facilitates memory copies into staging buffers when the app needs to blit tiles to the sparse texture. The second step creates a heap for the mapped tiles and a second heap for the staging buffers. Allocating buffers from a heap is more efficient because Metal won't perform expensive state tracking to avoid data hazards. The sparse texture manager performs its own heap management because only the sparse texture, in the grander scheme, needs to have data-hazard tracking. The third step maps the *mipmap tail*, the highest mipmap levels that fit inside one memory block. Then the texture manager blits the bottom mipmap tail into the sparse texture to ensure that all tiles contain a minimal amount of texture data. The final step creates the access counter buffer for all frames in flight. Lastly, the app updates the residency buffer to tell Metal which tiles are resident.

```
_numTilesToDiscardFromLRU = 0;
_sparseTextureBacking = [[AAPLStreamedTextureDataBacking alloc] initWithKTXPath:path
[self createHeaps:heapSize];
[self mapMipmapTails];
[self blitMipmapTails];
[self createAccessCountersBuffer];
[self updateResidencyBuffer];
```

At this point, the app has initialized the sparse texture, copied the bottom mipmap tails to GPU memory, and mapped the mipmap tails resident. The app may now use the sparse texture for rendering objects.

Render the scene

The app performs ordinary rendering tasks in `drawInMTKView:`, like updating animation variables and uniform buffers, creating a command buffer, and rendering the scene. The end of the following block of code shows an optional rendering pass that renders a quad in the lower-left of the screen. This quad shows a color-coded version of the residency buffer. You may disable this visualization by setting the preprocessor variable DEBUG_SPARSE_TEXTURE to 0.

```
id<MTLCommandBuffer> commandBuffer = [_commandQueue commandBuffer];
commandBuffer.label = @"Main render cmd buffer";

// Update the animation and uniform buffers for the sample.
[self updateAnimationAndBuffers];

// Begin the forward render pass.
[self drawScene:commandBuffer];

#if DEBUG_SPARSE_TEXTURE
// Draw a visualization of the sparse texture residency buffer.
[self drawDebugSparseTextureTiles:commandBuffer];
#endif

// Register the completion handler for the command buffer.
[commandBuffer addCompletedHandler:^(id<MTLCommandBuffer> _Nonnull cmdBuffer)
{
    dispatch_semaphore_signal(self->_inFlightSemaphore);
}];
[commandBuffer presentDrawable:_mtkView.currentDrawable];
[commandBuffer commit];
```

The app begins the update process after it commits the main command buffer. It asks the sparse texture class to update by querying the access counters and mapping and blitting tiles. The update can occur concurrently with the rendering thread using GCD. You may disable asynchronous processing by setting the preprocessor variable ASYNCHRONOUS_TEXTURE_UPDATES to 0.

```
#if ASYNCHRONOUS_TEXTURE_UPDATES
dispatch_async(_dispatch_queue, ^{
    // Process the sparse texture access counters, and map and blit tiles.
    [self->_sparseTexture update:self->_currentBufferIndex];
});
#else
// Process the sparse texture access counters, and map and blit tiles.
[_sparseTexture update:_currentBufferIndex];
#endif
```

Sample the sparse texture

The following code shows how the app draws the ground plane. It sets typical render states like pipeline state object, vertex and fragment buffers, and texture state. It also sets a fragment buffer, `_sparseTexture.residencyBuffer`, that the shader utilizes to sample the texture. And finally, it sets the sparse texture using `setFragmentTexture`.

```
[renderEncoder setCullMode:MTLCullModeNone];
[renderEncoder setRenderPipelineState:_forwardRenderPipelineState];
[renderEncoder setVertexBuffer:_sampleParamsBuffer[_currentBufferIndex] offset:0 at
[renderEncoder setFragmentBuffer:_sampleParamsBuffer[_currentBufferIndex] offset:0 a
[renderEncoder setFragmentBuffer:_sparseTexture.residencyBuffer offset:0 atIndex:AAP
[renderEncoder setVertexBuffer:_quadVerticesBuffer offset:0 atIndex:0];
[renderEncoder setFragmentTexture:_sparseTexture.sparseTexture atIndex:AAPLTextureIn
[renderEncoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0 vertexCount:6];
```

The shader code uses the function `sampleSparseTexture` to handle sampling from the sparse texture. Metal provides a `sparse_sample` function that returns a `sparse_color<half4>` object. This object has a `resident` member function that returns `false` for an unmapped tile region. If the tile is resident, `sampleSparseTexture` returns the sampled color. Otherwise, it uses the residency buffer to determine the best MIP level for each mapped tile. Then it resamples the texture with the `min_lod_clamp` argument to ensure that unmapped tile regions aren't accessed.

```
/// Sample the sparse texture and return a lower mipmap level if the tile isn't resi
half4 sampleSparseTexture(texture2d<half, access::sample> sparseTexture,
                          float2 texCoord,
                          float2 sparseTextureSizeInTiles,
                          const device char* residencyBuffer)
{
    constexpr sampler linearSampler(mip_filter::linear,
                                    mag_filter::linear,
                                    min_filter::linear,
                                    s_address::clamp_to_edge,
                                    t_address::clamp_to_edge);
    // The `sparse_sample` function returns a `sparse_color` type to safely sample t
    sparse_color<half4> sparseColor = sparseTexture.sparse_sample(linearSampler, tex
    half4 baseColor = half4(0.h);
    // The `resident` function returns `true` if the sampled region is mapped.
    if (sparseColor.resident())
    {
        baseColor = sparseColor.value();
    }
    else
    {
        float residentBufferMipmap = getResidencyBufferMipmap(residencyBuffer, spars
        // `min_lod_clamp` restricts the minimum mipmap level that the shader can sa
        baseColor = sparseTexture.sample(linearSampler, texCoord, min_lod_clamp(resi
    }
    return baseColor;
}
```

The residency buffer is a two-dimensional data structure that stores the best MIP level for each mapped tile. The function `getResidencyBufferMipmap` takes the input texture coordinates and converts them to tile coordinates `readX` and `readY`. The shader then indexes the residency buffer and returns the best MIP level.

```
/// Return the minimum mipmap level available in the sparse texture.
float getResidencyBufferMipmap(const device char* residencyBuffer,
                               float2 sparseTextureSizeInTiles,
                               float2 texCoord)
{
    // Transform the UV coordinate from a pixel coordinate to a tile coordinate.
    ushort readX = (ushort) (clamp(texCoord.x, 0.f, 0.99f) * sparseTextureSizeInTile
    ushort readY = (ushort) (clamp(texCoord.y, 0.f, 0.99f) * sparseTextureSizeInTile
    ushort index = readX + (readY * (ushort)sparseTextureSizeInTiles.x);
```

```
    ushort val = residencyBuffer[index];
    return (float)val;
}
```

While the fragment stage is running, the GPU records the number of texture accesses by the shader. The app analyzes this buffer to stream and map new regions of texture data that aren't resident.

The following figure shows an example of how the tiles sample resident parent tiles if a requested tile isn't resident. The green tiles show a tile that the shader accessed and was resident. The red tiles show a tile that the shader accessed, but had to fall back to a lower MIP level. The app detects a tile it needs to map when the access counter is nonzero and the corresponding residency buffer is zero.

Update the sparse texture

The following figure shows how the update process decides when to map or unmap tiles. For every resident tile that the shader accessed, the tile moves to the front of the least-recently used (LRU) cache, a data structure that combines a linked list and an unordered map. The `processAccessCounters` method creates map requests for the accessed nonresident tile and its nonresident parent tiles. The parent tiles must form a chain from the bottom mipmap tail to the highest level tile. The update process checks for any dependencies and doesn't create unmap requests for required parent tiles. And if the heap doesn't have enough memory available, then `discardTilesFromLRU` unmaps unnecessary tiles to make room.

To summarize, the `AAPLSparseTexture:update:` method calls four functions to update the texture:

- `updateAccessCountersBuffer` uses a blit encoder to get the access counters.
- `processAccessCounters` examines the access counter buffer to determine the tiles to map or unmap.
- `discardTilesFromLRU` uses an LRU cache to manage the sparse texture heap and determine the tiles to discard.
- `mapAndBlitTiles` maps tiles that need residency and blits them into the sparse texture.

The remaining sections cover these methods in more detail.

Update the access counter buffers

The sparse texture class uses the `getTextureAccessCounters(_:region:mipLevel:slice:resetCounters:countersBuffer:countersBufferOffset:)` API to copy and reset the counters for the sparse texture. It requests Metal to copy the data from each MIP level into the `_accessCountersBuffer`. The initialization step precalculated the offsets into this buffer, and the app can reference them from the `_accessCountersMipmapOffsets` array.

```
for (NSUInteger mipmap = 0; mipmap < _sparseTexture.firstMipmapInTail; ++mipmap)
{
    NSUInteger accessCountersOffset = _accessCountersMipmapOffsets[mipmap];
    MTLRegion pixelRegion           = [_sparseTextureBacking calculateMipmapRegion:m
    MTLRegion tileRegion;

    [_device convertSparsePixelRegions:&pixelRegion
                         toTileRegions:&tileRegion
                          withTileSize:_tileSize
                         alignmentMode:MTLSparseTextureRegionAlignmentModeOutward
                            numRegions:1];

    [encoder getTextureAccessCounters:_sparseTexture region:tileRegion mipLevel:mipm
                         resetCounters:YES
                        countersBuffer:_accessCountersBuffer[frameIndex]
                  countersBufferOffset:sizeof(uint) * accessCountersOffset];
}
```

Process the access counter buffers

When the sparse texture class examines the access counters buffer, each entry contains the number of times the shader accessed each MIP region. A value of zero means that the tile wasn't referenced at all in the last frame. Since there can be several frames in flight, the app ensures that tiles aren't unmapped prematurely. To manage this, the sparse texture class uses a simple data structure `TextureTile`:

```
struct TextureTile
{
    /// The x and y components represent the tile origin in the tile coordinate (not
    /// The z component represents the mipmap level to which this tile belongs.
    MTLOrigin origin;

    /// The state describes the status of the tile in terms of whether it's unmapped
    /// or if it's undergoing the process of mapping or unmapping.
    TileState state;
```

```
    /// The frames count variable makes sure a tile used by a previous frame isn't u
    int8_t framesCount;
};
```

The sparse texture class categorizes texture tiles in one of five states: unmapped, mapped, queue for mapping, queue for unmapping, or stored in the LRU cache. When examining each counter for all tiles, the sparse texture manager applies the following actions:

- Queue an accessed tile that's unmapped for mapping.
- Store a mapped and unaccessed tile in the LRU cache.
- Queue an unaccessed tile in the LRU cache for unmapping.
- Change an accessed tile in the LRU cache back to a mapped state.
- Do nothing if the accessed tile is mapped.

The `SparseTexture:newMapTileRequest:` method adds the tile to a list of tiles to map. The helper function `setTextureTileRefCounterParent` ensures that parent tiles are properly reference counted. Resident parent tiles may depend on tiles in lower mipmap levels, so the sparse texture class doesn't put them in the LRU cache. The following code shows the logic of putting tiles into the LRU cache.

```
// Only consider putting a tile in the LRU cache if it's mapped and isn't needed
// by a parent tile in its mipmap chain.
bool isTileNotMapped = (tile->state != TileState::TileStateMapped);
bool isTileUsedByParentTile = (mipmap > 0 && (_countRefParentTiles[mipmap - 1][tile]

if (isTileNotMapped || isTileUsedByParentTile)
{
    continue;
}

// Only put a texture tile in the LRU cache when the GPU hasn't sampled it
// in a recent frame.
tile->framesCount = std::max(0, tile->framesCount - 1);
if (tile->framesCount <= 0)
{
    tile->state = TileState::TileStateStoredInLRUCache;
    _notUsedMappedTilesLRUCache.put(tile);
    [self setTextureTileRefCounterParent:tile higherMipmapQuality:NO];
}
```

Discard tiles from the LRU cache

The app uses a heap of textures to manage the mapped tiles in the sparse texture. If there's no memory available to map nonresident tiles, then the sparse texture class discards older tiles. It uses an LRU cache to prioritize tiles to discard. The `AAPLPointerLRUCache` class manages a `std::list` and `std::unordered_map` to track mapped tile pointers. When the manager retrieves a pointer with `AAPLPointerLRUCache::get`, it moves the tile to the front of the cache. When the manager discards a tile and the cache is full, `discardLeastRecentlyUsed` removes the last entry in the cache. The app tracks the number of tiles that need discarding and creates unmap requests in the following code:

```
NSUInteger index = 0;
for (; index < _numTilesToDiscardFromLRU; ++index)
{
    TextureTile* tile = _notUsedMappedTilesLRUCache.discardLeastRecentlyUsed();
    if (!tile)
    {
        break;
    }
    [self newUnmapTileRequest:tile];
}
```

This completes the process to get the access counter buffers and create the map and unmap requests. The next step is to map and blit tiles.

Map and blit tiles

The app stores a list of mapping and unmapping requests that the `mapAndBlitTiles` method encodes using a *resource state command encoder*. The `updateTileMappingMode` method converts the sparse pixel regions to tile regions and then updates the texture mapping to reflect the highest mapped MIP level.

```
id<MTLResourceStateCommandEncoder> rsEncoder = [cmdBuffer resourceStateCommandEncode
rsEncoder.label = @"Tile mapping resource state encoder";

for (TextureTile* tile : unmapTilesRequest)
{
    [self updateTileMappingMode:tile mappingMode:MTLSparseTextureMappingModeUnmap or
    tile->state = TileState::TileStateUnmapped;
}
```

```
for (const auto tile: mapTilesRequest)
{
    [self updateTileMappingMode:tile
                    mappingMode:MTLSparseTextureMappingModeMap
                      onEncoder:rsEncoder];
}

[rsEncoder endEncoding];
```

While the resource state encoder is processing, the app starts streaming the tiles from the KTX file and blits them into the texture. The sparse texture manager iterates over new tile requests and calls `streamTileToStagingBuffer` to allocate staging buffers from the heap. The manager copies the texture from the file to the staging buffer and uses a blit encoder to write it to the sparse texture.

```
// Stream the tiles from the source texture file into the sparse texture heap tiles.
for (const auto& tile: mapTilesRequest)
{
    id<MTLBuffer> tempStreamingBuffer = [self streamTileToStagingBuffer:tile];

    // `blocksWide` holds the number of blocks of compressed pixels spanning the wid
    NSUInteger blocksWide       = calculateBlocksWidth(_tileSize.width, _sparseTextu
    NSUInteger bytesPerRow      = blocksWide * _sparseTextureBacking.bytesPerBlock;
    MTLOrigin destinationOrigin = MTLOriginMake(tile->origin.x * _tileSize.width,
                                                tile->origin.y * _tileSize.height,
                                                0);

    [blitEncoder copyFromBuffer:tempStreamingBuffer
                   sourceOffset:0
              sourceBytesPerRow:bytesPerRow
            sourceBytesPerImage:0
                     sourceSize:_tileSize
                      toTexture:_sparseTexture
               destinationSlice:0
               destinationLevel:tile->origin.z
              destinationOrigin:destinationOrigin];
}
```

The last step is to wait until the blit command encoder is finished. During this time, the `mapAndBlitTiles` method updates the residency buffer and parent reference counts. This function waits to update the residency buffer until after the blits have finished, so the shader doesn't access data that hasn't finished mapping. The following code shows this process:

```
// Finish the tile requests to update the residency information.
for (auto& tile: mapTilesRequest)
{
    tile->state = TileState::TileStateMapped;
    [self updateTextureEntryResidency:tile minMipmapFlag:YES];
    [self setTextureTileRefCounterParent:tile higherMipmapQuality:YES];
}
```

Once the resource state encoder maps the tiles, the blit encoder copies the texture data and the app updates the residency buffer, the process repeats for each frame. The app code renders a quad to the screen using the residency buffer to show the highest MIP levels available to visualize the sparse texture tile residency. You may set the `USE_SMALL_SPARSE_TEXTURE_HEAP` preprocessor variable to 1 to see how mapping and unmapping occurs more frequently when the heap size is smaller.

See Also

Textures

{} Processing a texture in a compute function

Create textures by running copy and dispatch commands in a compute pass on a GPU.

{} Reading pixel data from a drawable texture

Access texture data from the CPU by copying it to a buffer.

{} Creating and sampling textures

Load image data into a texture and apply it to a quadrangle.