VirtualMemory

Abstract

一种类似LOD(level of detail)的方法，主要是适用于使用的纹理体积过于庞大显存没有办法存储的时候使用。但它并不是一种扩大存储空间的方法，而是一种优化空间结构的方法，即便它看起来像虚拟内存的架构。

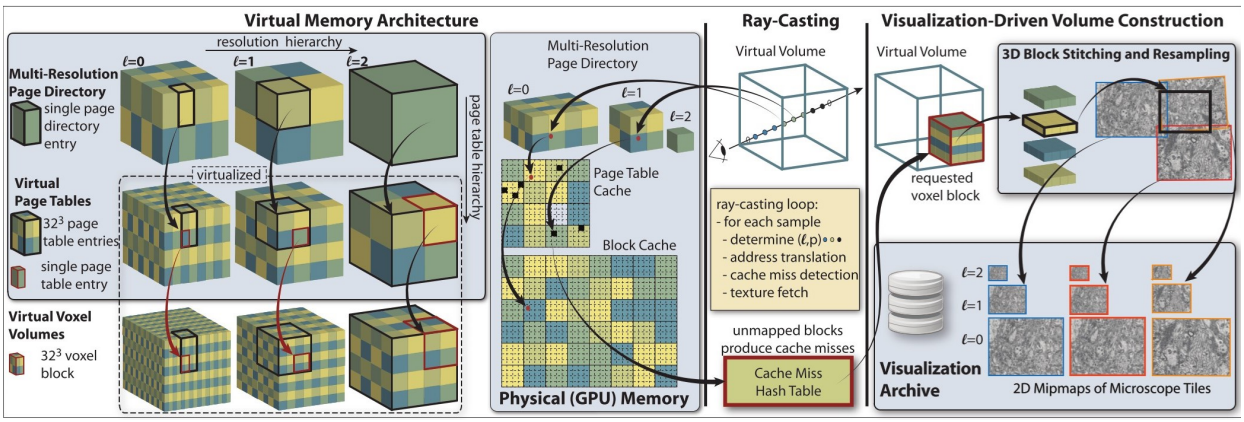
Theory

理论介绍，介绍和解释VM(Virtual Memory)方法。

Base Theory

它有两个主要的思想，第一个是关于多分辨率，第二个是关于纹理内存空间结构的。

- 多分辨率：对于一个庞大的纹理，如果想要在同一时刻使用所有的数据，例如在一次DrawCall中使用，除非是基于硬件的支持或者底层的支持，例如Windows系统中的虚拟内存系统，不然是没有办法同时使用的。但是这样的情况是存在的，例如在体渲染中我需要渲染整个体纹理。因此，基于LOD的想法，在构造纹理的空间(内存空间)结构中，考虑了对多种分辨率的支持。
- 纹理内存空间结构，和普通的内存结构是非常类似的。将结构分成目录(Directory)，页(Page)，块(Block)这三种类型。对于目录，存储页的索引(Page Entry)，对于页，存储块(也可以是页)的索引(Block Entry)，而对于块来说，则存储具体的数据。但是对于目录和页来说，存储的每一个索引都附带一个属性，所代表的数据是否被映射，换句话说则是数据是否在内存中。通过这样的一个结构，我们可以访问到存储在内存中的数据，同样也知道哪些数据没有在内存中。



Representation

在基本思想中提到过，整个结构分为三种类型，目录，页，块。也提到过其对于多分辨率的支持。

基于两种思想，目录的结构会稍微有一些改变。对于目录来说，它有了一个额外的属性，分辨率等级(resolution level)。

分辨率等级区分了我们要访问的纹理的精细度，我们是需要访问没有进行任何处理的纹理，还是访问经过压缩，或者经过降采样后得到的纹理。可以参考mipmap，其对于一块纹理存储了多个大小不同的纹理，而分辨率等级则决定了我们要使用哪块大小的纹理。

那么对于目录来说，其可以通过一个分辨率等级，以及一个需要使用的纹理坐标来确定我们需要使用的页的索引，我们可以通过这个页的索引来访问页内的信息。

对于页和块来说，其结构不同于目录。我们不可能将所有的页和块都存储在内存中，这样肯定会超过内存容量。因此我们只会存储一部分的页和块放在内存中，其结构和内存池类似。这也是为什么对于页和块的索引需要附加上一个"是否被映射"的属性，对于被映射的页和块，我们可以通过索引来访问其数据。

Framework

整个的框架也是非常简单的。

```
...

access stage loop:
    get (resolution, position)
    query page entry from directory using (resolution, position)
    if not mapped : report cache miss
    query block entry from page using (resolution, position)
    if not mapped : report cache miss
    query data from block using  (resolution, position)
...
```

Memory Object

对于页和块来说，我们需要维护一个LRU(Least Recently Used)系统，因此我们不妨设计一个LRU基类用于内存池中的元素，并提供接口访问元素以及分配元素的内存空间(在接口中处理关于LRU系统的细节)。

对于目录，页来说，其存储的是索引以及是否被映射的信息，我们可以构建一个虚拟的链接(virtual link)来记录索引和映射信息。对于索引，我们则是使用一个虚拟的地址

(virtual address)来表达。这里通常是一个整型的三维坐标(对于三维纹理来说)。那么也就是说，目录和页都是一组虚拟链接的集合，通常我们使用三维纹理的形式维护，但是其本质还是一个一维的集合。

Address Translation

对于一个访问操作，其输入是$[resolution, position]$ 。其中:

$$resolution \in [0, max]$$

并且$position$ 是一个三维向量

$$position.xyz \in [0, 1]$$

我们想要访问的数据其所在的位置我们使用的是一个三维向量来表达，或者说它实际上是一个纹理坐标。我们需要通过这个纹理坐标，来计算出访问的数据其所属的块。其方法也类似于纹理采样一般，只不过少了一些步骤。

- 首先我们通过$base[resolution]$ 来得到当前分辨率下使用的目录的偏移位置
- 然后我们通过将位置和当前分辨率下使用的目录的大小各分量相乘，再加上偏移位置，那么我们就得到了数据所在的页它的虚拟链接在目录中的位置，从而我们得到了页的虚拟链接
- 通过页的虚拟链接，我们将位置和块的个数相乘，并将其对页的大小取模，那么我们就得到了数据所在的块它的虚拟链接在页中的位置，从而我们得到了块的虚拟链接
- 通过块的虚拟链接，我们将位置和体素个数相乘，并将其对块的大小取模，那么我们就得到了数据在块中的位置，从而我们得访问到了数据。

其伪代码为

```
directory entry <- get entry [base[resolution].xyz + position * directory siz
page entry <- get entry [(position * block count) % page size]
block data <- get data [(position * voxel count) % block size]
```

Cache Miss

访问数据代表着我们需要这个数据，存在可能性我们访问的数据没有被映射，称之为缓存未命中(Cache Miss)。因此我们需要一个方法来处理这样的情况，即汇报未命中的情况，然后将其读入到内存中去。这个和CPU中寄存器有一点类似。

具体来说，我们为页和块维护一个LRU系统，然后在每次间隔，处理所有汇报的未命中事件，对于未命中事件，将其对应的块的读入到内存中去，替换掉LRU系统中最末尾的

块，如果块所处的页也没有被映射，那么同样将新的块所处的页替换掉LRU系统中最末尾的页。同时也维护一个内存中块在上一次访问阶段的使用状态，如果块被使用，则将其放到LRU系统的最前面，使得我们优先将未使用过的块重新分配给未命中事件所载入的块。

这样的话，就保证了在内存中的块都是一定在上次访问阶段需要的块，而那些不需要的块，则只有在需要的块数量比在内存中维护的块少的时候，才会在内存中保留下来。

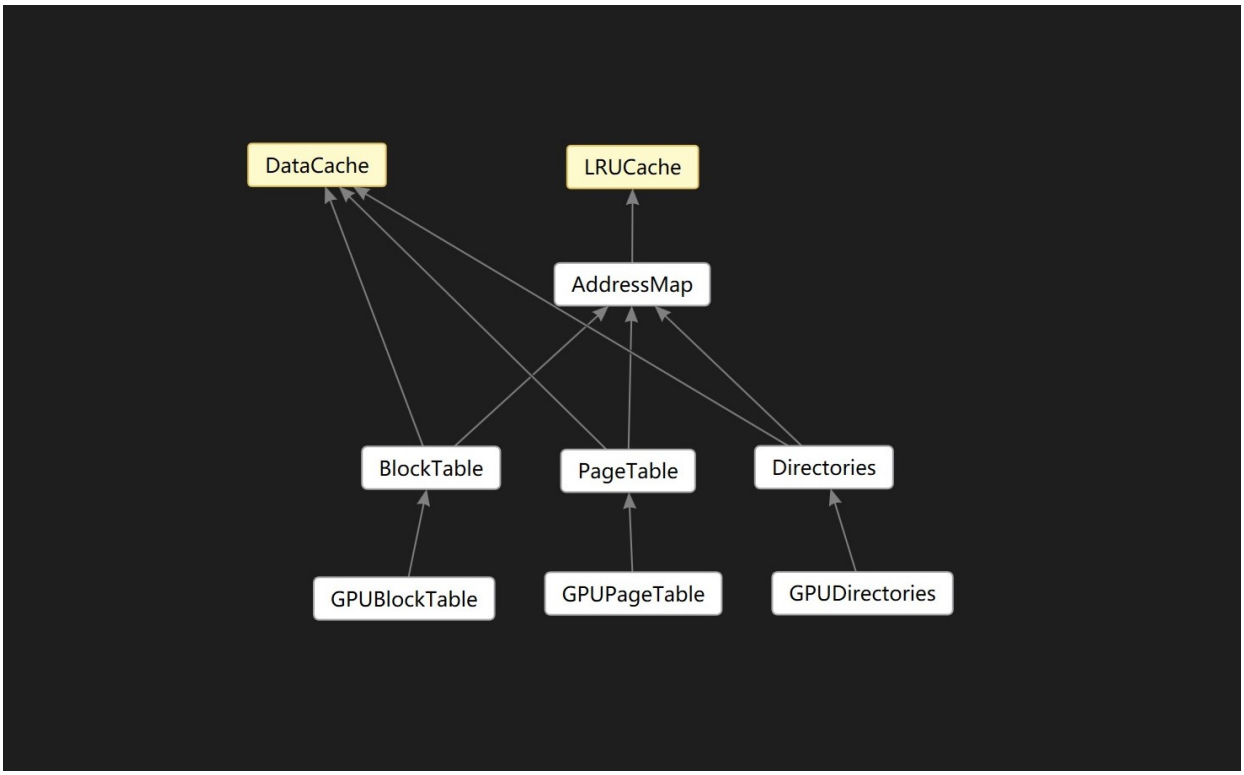
```
for each block in the memory
    if not mapped : continue
    LRU system access block

for each cache miss in cache miss table
    if mapped : continue
    load block from disk
    add block to virtual memory object(directory, page, block)
```

Implement

我们以3D纹理在体渲染中的使用作为例子。

Interface Architecture



DataCache, LRUCache是最基础的块，其类似一个三维数组一般，但是对于LRUCache来说，其内部的数据由哈希表和链表维护，用于维护LRU系统。

AddressMap将LRUCache整合起来，并且设计出方便与Memory Object继承的接口。

BlockTable， PageTable， Directories等继承自AddressMap，因此也是一个块，但是其元素则是虚拟链接。

需要注意的是他们只是所有的目录，块，页的集合，因此我们还需要在内部维护一个内存池，存储所有的页，块，其元素则显然继承自DataCache。

而前缀 `GPU` 则代表着其维护的数据是存储在GPU内存中的，因此其内部的内存池则变成了纹理(存储的信息类型不变)。但是由于LRU系统在Shader内部是不好实现的，因此实际上我们仍然需要在目录和页中维护一个内存池放在CPU内存中，用于维护LRU系统，其数据和GPU内存中的保持一致。而对于块来说，为了节约空间以及提高效率，我们并不在CPU内存中存储实际上的块数据，而是存储到GPU内存中。

Interface Design

为了方便实现，我们将LRU整合至AddressMap的同时，也为AddressMap设计了一组接口。

- `mallocAddress`：分配一个地址出来，如果空间满了，则分配的是LRU系统中最末尾的元素，否则则是任意未被使用的元素。
- `setAddress`：设置一个地址的元素，将会触发LRU系统，即这个地址的元素将会放入LRU的头部。
- `getAddress`：获取一个地址的元素，将会触发LRU系统，即这个地址的元素将会放入LRU的头部。

而对于目录，页，块等，我们同样页设计了一组接口。

- `mapAddress`：将一个块映射到内存中去。
- `clearUpAddress`：将地址代表的页或者块清楚，需要注意其关系的维护。

Cache Miss Report

显然，框架中的访问阶段在体渲染中则是具体的渲染代码，即Ray-Cast阶段。对于每一条射线，对其进行等距离采样。对于每一个采样点，通过目录，页，块的纹理来访问数据，如果块没有被映射的话，那么就汇报缓存未命中。

我们使用哈希表(三维纹理实现)来记录未命中的信息。首先我们将整个渲染的范围分块(通常是分成64x64块)，对于每一个块维护一个哈希表。对于任意一条光线，我们将未命中的块的编号放入到所属的哈希表内。

需要注意的是，对于每个哈希表，我们限制了未命中事件的次数，其主要是为了性能的稳定。同时，我们可以限制每一条光线最多在一次Ray-Cast的时候汇报缓存未命中的次数，这样可以使得汇报未命中的Ray-Cast较为平均。

而对于内存中纹理的使用情况的记录，我们同样使用一个三维纹理记录。其位于块集合的坐标就是其纹理坐标。通过纹理坐标我们就可以存储其是否被使用的信息。

当访问阶段结束后，我们就需要解决所有的未命中以及更新块的使用状态。首先我们需要更新块的使用状态。然后再解决未命中事件。因此我们的纹理需要在CPU中读取。

最后需要注意的是，由于Shader是并行的，因此具体的执行顺序位置，所以在纹理中的读写操作建议使用atomic操作进行。

Resolution Detection

之前提到过，我们使用的是支持多分辨率下的方法。因此在体渲染中，我们是需要去确定我们使用的分辨率的等级(每个等级对应具体的分辨率缩放)。

这里我们使用的方法是计算出摄像机的平截头体，然后对体渲染的物体进行裁剪，然后得到一个多面体，通过叉积计算出其体积，从而得到其可见部分的体积比。再通过体积比，去一一比较在每个分辨率等级下，其所需要的空间大小是否能够被完全加载到内存中去，从而选择出最合适的分辨率等级。

但是由于这样的方法计算的是体积，而不是使用子块去拟合图形，因此得到的数据是有一定的偏差的。但是这样的性能相比更高。

Detail && Notice && Optimization

- 对于每个Memory Object，实际上还需要维护一个虚拟链接的反向链接。用来方便处理当前页或者块失效的时候，将指向这个页和块的链接设为未映射状态。
- 一定不要对每个页和块去维护LRU系统，而是对它们的集合维护。
- 访问的数据的坐标通常是一个浮点类型，需要注意其对大小的乘积的范围是[0, size]的，因此需要注意越界处理，尤其是在Shader代码中的。
- 在纹理采样中，可以测试下上次采样点的块是否和当前采样点的块一样，这样可以减少纹理采样次数和计算。
- 可以不仅仅只加入是否映射的属性，还可以加入是否为空的属性，来优化采样的性能。对于空的块或者页，直接跳过不采样。
- 可以在CPU中维护一个更大版本的VM，当需要的时候先从CPU中查询是否存在，再从硬盘中查询。

Sparse Leap

可以使用Sparse Leap来跳过空体素。具体的内容点击链接。

在VM中的实现，我们在处理缓存未命中的时候来维护Sparse Leap的八叉树。当有一个块被载入的时候，将这个块进行处理，分割成对应八叉树中对应大小的块，同时确定这个块的类型，然后插入八叉树中。当一个块被移除的时候，同样也可以将其对应八叉树的块移除，从而动态维护了八叉树的大小。

Source Code

请点击链接

Reference

[0] : M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a VisualizationDriven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012.

[1] : Markus Hadwiger, Ali K. Al-Awami, Johanna Beyer, Marco Agus, and Hanspeter Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974-983, 2018.