RAGE 来了

－－－关于 ID TECH 5 MEGATEXTURE 的一些技术信息更新

H3D 姚勇

信息

很久很久以前，ID TECH 5 有风声时，我曾经查到一篇论文。就当时 id software 公司的 MEGATEXTURE 技术做了一些介绍。是关于 Clipmap 的。 id software 采用此技术制作了《Quake War》之后，公司很快更新了 ID TECH 5（MEGATEXTURE v2.0），采用了类似 Virtual Texture 的技术，做到了地形和场景物体都使用一张超大全细节贴图做 texture streaming。 如今 RAGE 已经发行。作为 id 技术 fan 买了 PS3 正版之后，很失望..... 游戏体验，尤其在贴图绘制上很糟糕。后来发现在 PC 上的体验其实不错。贴图 loading 的感觉较小。原因大概因为 id software 在 PS3 上优化的不够或者 PS3 架构就不适合 id tech5 。

Clipmap 是用于地形渲染的。 优点是绘制超大细节地形纹理，不受显卡显存限制，而且不需要过多预处理过程。 在使用 ID TECH 5 技术的《RAGE》中，不再使用 clipmap。而是使用一种基于 virtual texture 的技术 。关于 virtual texture 技术雏形，最早源自上个世纪一家名叫 3DLABS 的显卡公司 。那时是实现在硬件中的。这点与 clipmap 很类似。

id software 的纯软件技术实现的灵感则来自一篇 04 年的叫《Unified Texture Management for Arbitrary Meshes》论文。最终 id 实现的这个技术应该比 3DLABS 的要远远复杂得多。并且包含非常多的技术实现与优化细节。以及全套美术制作流水线。一个引擎和游戏开发了 6 年多，复杂是显而易见。

在 08 年 GDC 上一位技术爱好者凭着一些隐约的 ID TECH 5 线索，以及有限的 paper，自己实现了一套类似 megatexture 的技术。09 年 id software 做技术介绍时，证明这位仁兄的

实现和 id 的很像。同时 08 年 crytek 公司也介绍了他们在 cryengine 中实现的 virtual texture 的想法。

具体 virtual texture 的技术，要做成产品级应用，实现非常复杂。是一个工程难度很高的技术。本人没做过。只简单介绍一下。水平有限，不保证信息的完全准确。

介绍

一，目的

为何需要这个技术。简单讲原因有二。

第一，显存有限。次世代游戏动辄就要几百兆显存来支持精细的各种贴图（ diffuse, specular, 各种 mask，各种 bump，各种 env, decal, 等等等等）。随着游戏制作规模增大，要更多的贴图来满足眼球。只有更精制，没有最精制。关卡没有最大，只有更大。 但是显存有限。

第二，普通制作游戏场景中的各种景物，贴图重复使用是非常关键的事情。因为一个 3D 游戏，关卡足够大，虚拟现实沉浸感才强。细节物体要多。只有更多，没有最多。更大的场景更多的景物，需要更多的贴图是一方面，另外一方面，需要美术对贴图的复用考虑更多，在制作上也需要大量精力去调整 UV 和优化。

为了节省贴图，一般美术制作还需要使用贴图 Tilling。就是一张贴图重复排列开来，铺在面积很大的物体表面上。这样显得很假。这需要美工花极大精力去制作既要重复排列又不能显得重复性太明显的贴图。美工很多工作并不是象绘制图画一样，而是更像个砌墙工，竭力想用重复的材料堆砌出不一样的效果。生产力不够高。

二，实现

virtual texture 想法很直接：假设屏幕分辨率 1680x1050 。 那么这个屏幕可以显示的贴图图素（texel）数据，最多不过 1680x1050x3 字节。 如果我们把屏幕上要显示的景物，面向屏幕的可见部分，精确的制作一张贴图，那么这个贴图最大就需要 1680x1050x3 = 5MB 。前提是屏幕上显示的景物，背对屏幕的地方和被遮挡的地方都没有贴图。 如果我们能够实时创建这么一张贴图来显示场景，我们显卡的贴图用显存只用 5MB 就完全够了。

这种极端的情况用现有的硬件实现起来恐怕不太现实。退而求其次：我们有一些 cache 做冗余，景物有一些互相遮挡的绘制，就算 6 张屏幕大小的贴图，重叠遮挡绘制 3 次，每个像素 4 字节，那么在这种很差的情况下，我们也就需要 120MB 贴图。在次世代游戏显卡上，这个要求不算过分。

所以自然而然产生了一个想法。 那就是把这个游戏所有要用到的贴图全部放在一张超大的贴图上。在绘制每一帧时，只把屏幕上要显示出来的最高精细度贴图从硬盘上装载到显卡显存，绘制出来就可以了。 下一帧玩家镜头移动时，再去装载新景物的高精度贴图。如果不装载高精度贴图，景物可以用很低精度的贴图显糊弄一下(Mipmap Level 很高)。

这个过程如果很快，并且只有离玩家较远的新显示景物去装载高精度贴图。那么就可以用一张没有容量限制的超大的贴图（硬盘限制），在显存有限的硬件上，绘制一帧中眼力所

及的整个世界。并且这个世界的精度不会太受影响。换句话，游戏世界的贴图制作不用考虑显存大小。

并且，美术制作这个世界时，只用很直接的 2 步。1，做网格。2，给这个网格绘制唯一的贴图。不用考虑贴图复用。

ID TECH5 的 megatexture 技术就是要实现这个目标。

1，预处理

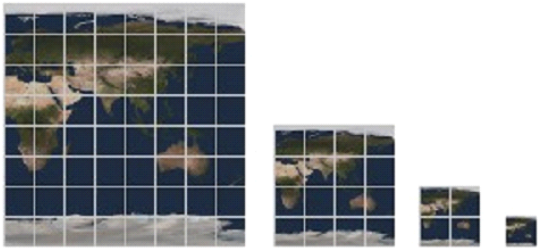
1.1 贴图预处理

在渲染之前，数据制作阶段时，先需要做一些预处理。最重要的一个处理，就是把世界中用到的所有贴图。全部先放入一张巨大的贴图。这也就是 megatexture 了。我们今后把这张能够绘制整个世界的唯一贴图叫做：virtual texture 。这张贴图可以很大，比如边长 200 多万像素，即一张 4T 大小的贴图（当然游戏中没有必要这么大）。

接着，按照一个固定大小的尺寸(《RAGE》中，是 128x128)，把 virtual texture 切割成方块。这个方块，我们称作“页”（pages）。

在这里，对 mipmap 的处理有点特殊。所有贴图的 mipmap，每一层也都按照这个固定大小切成 page。如果一张贴图的某一层 mipmap 尺寸小于 page 的尺寸，就停止切割。然后把 mipmap 切割出来的 page 也放在 virtual texture 文件里。

以下是一张贴图和这张贴图的 mipmap，被切割为 page 的样子。从左到右依次是 mipmap level 0,1,2,3 。 这也是 virtual texture 文件中的内容。

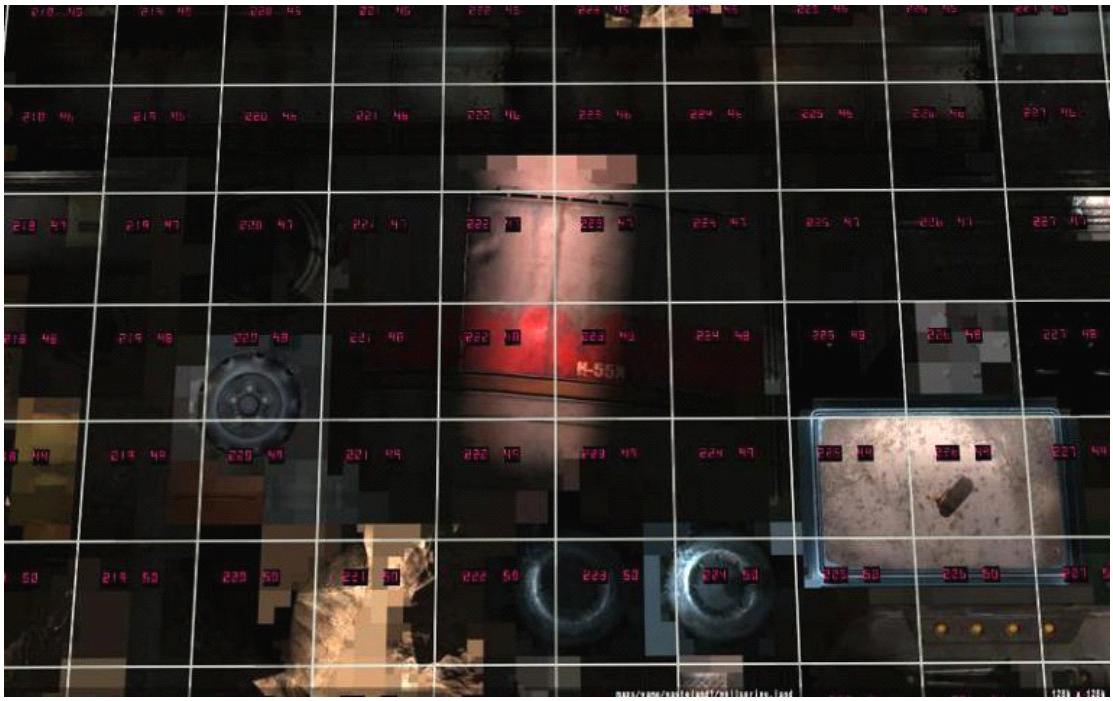


把这些 page 调入显存，再去显示场景。如果我们故意保留切割的痕迹，让我们看一下场景会是怎么样的：

在《RAGE》中，一张 virtual texture 是 128k x 128k 大小。这里我们说的 virtual texture 的尺寸，是最高精度的贴图尺寸。 一个 page 是 128 像素。 以下是《RAGE》中一个局部的 4 级 mipmap 在 virtual texture 中的样子。可以看到 4 张截图， 字体的 4 种颜色代表着 4 级 mipmap 。2 个数字是 page 在 virtual texture 中的全局唯一标号。

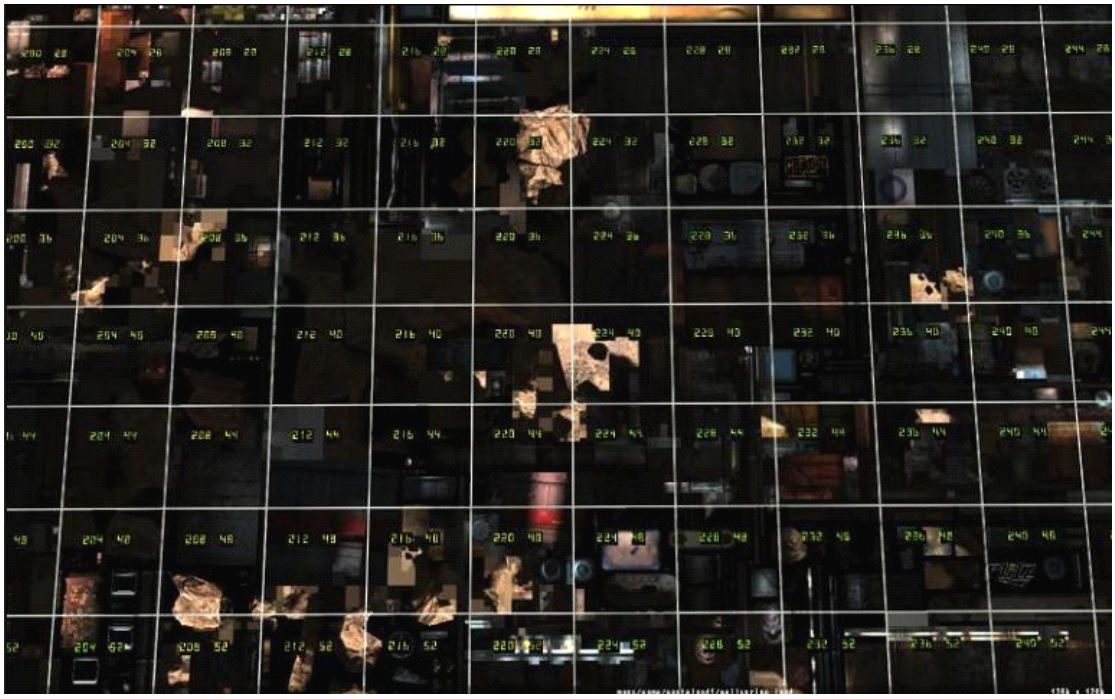
场景截图中，最靠近镜头的标号颜色，是 mipmap 精度最大的层。随着距离渐远，标号颜色也体现了使用精度更低的 mipmap。其中白色的格子表明是 page 的分界线。

LEVEL0：

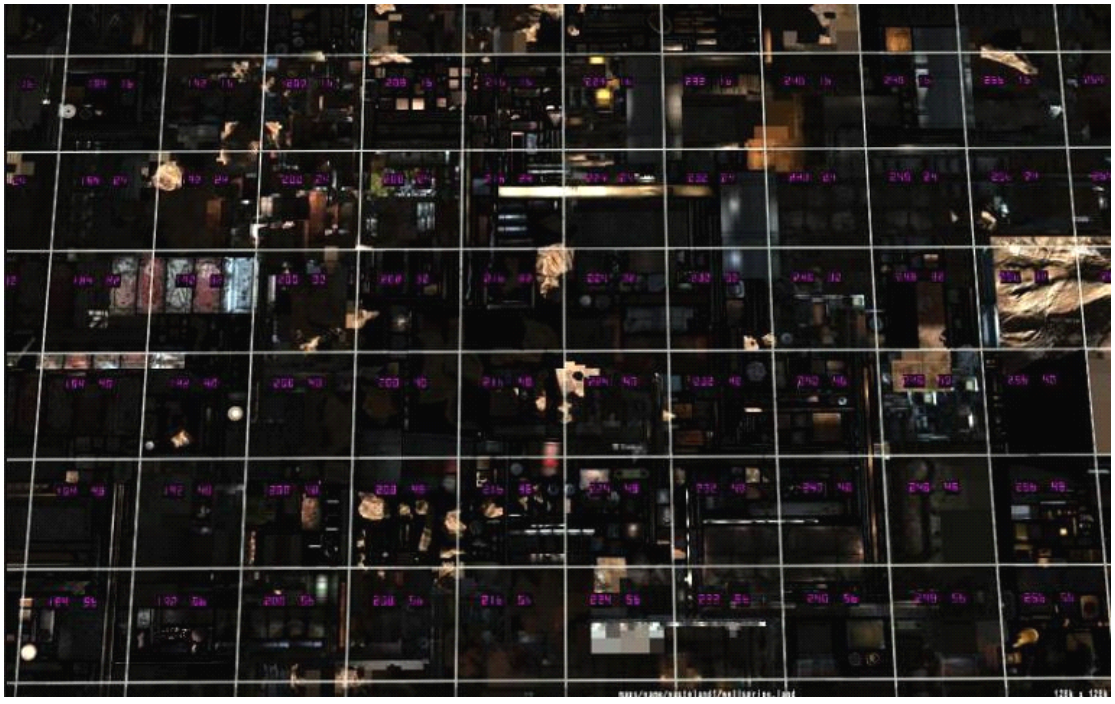


LEVEL 1:

LEVEL2:

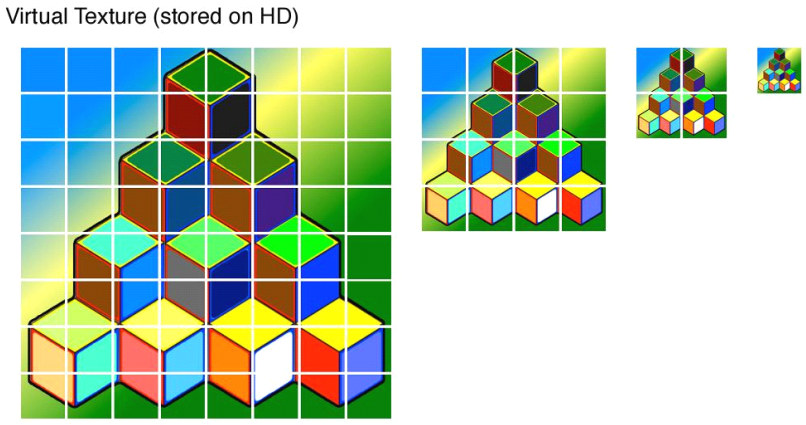


LEVEL3:



注意这里是对同一部位不同 mipmap 在 virtual textuer 上的截图。这些图都是放在 virtual texture 一个文件里的。

再给个例子。这个是一个 virtual texture 文件内部可能的一种组织形式：



1.2 几何预处理

把所有物体的贴图拼接在一起变为一整张 virtual texture（也就是 megatexture）后，世界中所有的物体都只使用 virtual texture 这么一张贴图。所以世界中物体的坐标都应该换算一下。

假设 virtual texture 是一张 131072 x 131072 大小的贴图。virtual texture 左下角贴图坐标为(0，0). 右上角贴图坐标为(1，1). 根据一个物体贴图在 virtual texture 中所在的相对位置，用很简单的换算即可得到这个物体新的贴图坐标。

比如一个物体有一张 512x256 的长方形贴图。位于 virtual texture 中从横向第 8088 个像素，纵向第 11901 个像素开始。这样物体一个顶点的贴图坐标 u,v (0<u<1 ,0<v<1)换算到 virtual texture 中，就是

u' = (u*512 + 8088)/ 131072 ，

v'= (v*256+11901)/131072

当贴图坐标 u,v<0 或者 u,v>1 时，说明使用了纹理 tilling。贴图重复排列。这时的处理，需要在 virtual texture 中重复复制物体所用到的贴图。 然后把贴图坐标重新换算到 0-1 之间。再进行 virtual texture 贴图坐标转换。

2，绘制

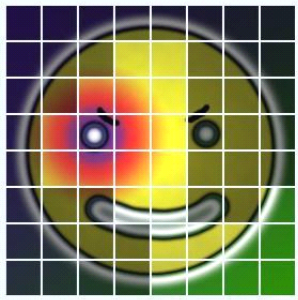
接下来绘制。我们做最简化情况的假设，有 3 步要走：

1，判断需要哪些 pages

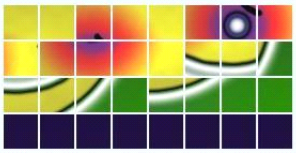
判断屏幕上需要绘制哪些贴图。以及屏幕像素上那个贴图用到的 mipmap 级别。因为世界所有贴图都以 page 为单位，拼在磁盘上的一张 megatexture 里。没法一次性调入显存。要判断出真的需要哪些贴图，以及那个贴图的 mipmap 级别。才会把需要的那部分 pages 装载。

2，装载

靠 streaming 把需要的贴图 pages 装载进来。放入显存的一张实际要使用的贴图（在 id software 的介绍中叫 physical page texture，在本篇文章中叫“显存贴图”。这张显存贴图，是由很多 pages 拼凑而成的。



假设这是屏幕，屏幕上亮的地方，是判断出要绘制的。其它暗的地方假设被其它物体挡住了



把要绘制部分需要的贴图，从virtual texture中以page为单位装载到“显存贴图”中。这就是“显存贴图”的样子

3，绘制场景物体。

绘制过程中，在 pixel shader 中，把屏幕上所有景物的贴图坐标重新换算一下。以便用显存贴图（贴图内容是从磁盘上靠 streaming 读出来的小块贴图（pages）拼凑而成）来正确绘制场景物体。因为场景中的物体在美术制作时，是按照普通展 UV 的方式制作的。

3，具体步骤

以下对上面说的 3 个步骤做具体介绍

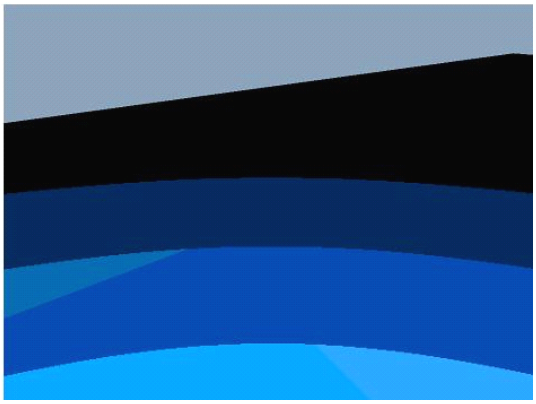
3.1 判断本帧绘制要用到哪些 page

page 中不仅仅记录了物体原始贴图的某个部分，同时这个原始贴图的 mipmap 也都切成块当作 page 储存在 virtual texture 中。所以这一步骤不仅要找到屏幕上物体用的贴图在 virtual texture 中的位置，还需要判断使用了哪个 mipmap 层，这样才能精确定位出屏幕上物体渲染需要的所有 pages。因为不同 mipmap 级别的 page 放在了文件不同地方。

用一个简单的实现办法概述这个过程：物体贴图坐标本身就说明了所用到的贴图在 virtual texture 中的位置。只要把物体贴图坐标绘制在屏幕上。再把屏幕内容取下来，逐点判断，就能够得到本帧到底需要哪些 page 来绘制。

详细一点的解释：因为所有物体只使用 virtual texture 这么一张贴图。那么物体贴图坐标肯定是全局唯一的（物体使用的贴图肯定分散在 virtual texture 的不同部分）。我们把物体贴图坐标当成顶点颜色，绘制一遍物体，那么屏幕上的像素就代表了这些物体的每个像素的贴图坐标（这里需要一点光栅化的知识。在绘制三角形时，顶点上的颜色作为填充三角形的颜色，被光栅化插值为三角形扫描线中每个像素的 RGB 值）。换句话，屏幕上绘制出来的就是绘制物体每个像素要用到的贴图坐标。同时在光栅化时，我们在 pixel shader 中还可以计算出每个像素用到贴图采样的 mipmap 层级数。（如果这里不懂可以先复习一下光栅化）

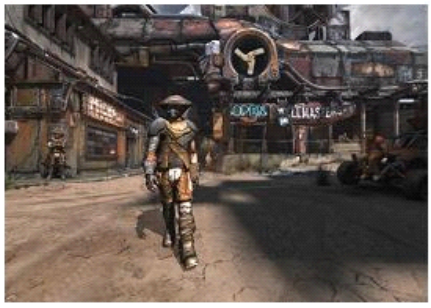
最后把这个结果输出为一张图片。由 CPU 去分析。 这里给出图片可能的样子：



环状不同颜色代表不同 mipmap。

这个东西在 id software 的介绍中，叫 Feedback Buffer。下面是《RAGE》中的样子

Color Buffer



Feedback Buffer

接下来就容易了，得到屏幕上的所有像素。逐一分析。每个像素的信息就是对应物体所用到贴图 page 的位置信息和 mipmap 级数。 这个级数也用来定位 page 的位置。

分析完毕，我们就有了当前屏幕上所有景物要用到的 pages 。我们把本帧需要用到的 page，叫“活跃 page”

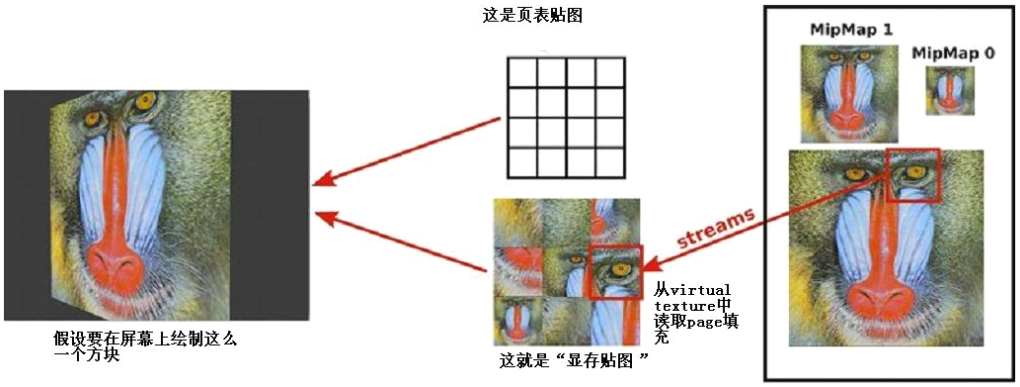
1680x1080 的屏幕上，像素个数是 1814400 个。 每个像素都检查一遍，CPU 基本就废掉，不用做其他事情了。 优化的办法是对屏幕 buffer 做 down sample 。也就是缩小屏幕 pbuffer 尺寸。只检查有限的像素。或者干脆直接在一个分辨率很小的 pbuffer 上绘制整个场景。诸如此类。 从图上看，RAGE 用的是一张分辨率比较低的 Feedback Buffer。

3.2 读入贴图 page

显存贴图(physical page texture)更新

知道要用到哪些 page 之后，就从磁盘读入进来。其中用到的各种 streaming 优化，贴图压缩解压缩，就不提了。

显卡中维护一张真正最后渲染要用到的贴图。我们叫它“显存贴图”或者“物理页贴图”（physical page texture）。这张贴图边长是 page 尺寸的整倍数。读入的 page 贴图方块，就一个个在 physical page texture 中拼接起来。（拼接的时候每个 page 周围还需要留条边。 大家都知道显卡最后纹理映射时，有个双线性插值的问题。而且随着物体离镜头越远，用的 mipmap 不同，对不同 mipmap page 周围留的边，还有粗细不同的问题。 总之这些零零碎碎的问题充斥着 megatexture 实现的所有步骤。）



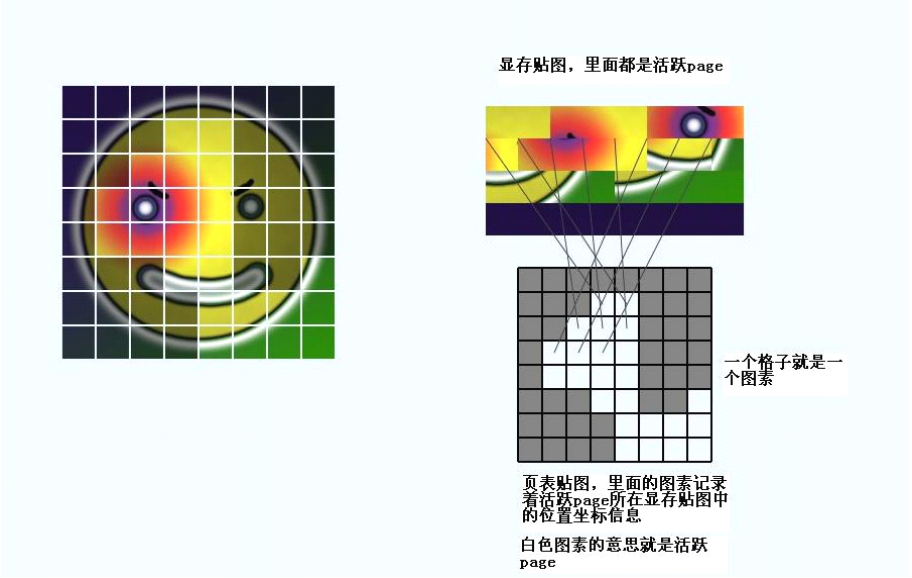
页表贴图(page table texture)更新

在这一步很重要的事情，还需要更新一张贴图。这张贴图叫 page table texture ———“页表贴图”。

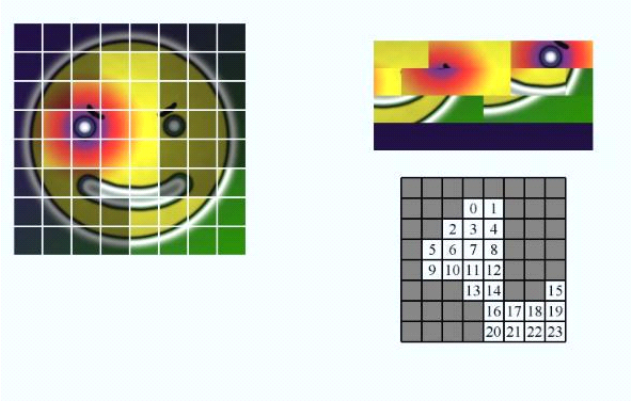
“页表贴图”大致的做法是这样。我们准备一张 virtual texture 的微缩版本。这个微缩版，有的 paper 叫 indirect texture，有的 paper 叫 page table texture 。 我们用中文，叫"页表贴图" 。页表贴图的每个图素，对应着 virtual texture 中相应位置的 page 。图素内容就是对应 page 在显存贴图（physical page texture）中的位置信息。比如：如果 virtual texture 最高精度那层有 1024x1024 个 page，那么页表贴图大小就是 1024 x 1024 个图素。

接着我们定义一下“活跃 page”。活跃 page 的意思，是说凡是本帧需要绘制，并从 virtual texture 被拷贝到显存贴图(physical page texture)中的 page，都要被标记为活跃 page。

页表贴图(page table texture)需要每帧更新，每个活跃 page 对应在页表贴图(page table texture)位置的图素内容，要记录这个活跃 page 在显存贴图(physical page texture)中的位置信息。 如图：



得到结果是：



因为在屏幕上的场景，会有 mipmap。用到不同 mipmap page 的信息，也记录在页表贴图的 mipmap 中。如图

显存贴图(physical page texture)用 page 填充好之后，更新完 page table teture 就可以用这两张贴图最后渲染了。

3.3，渲染

到这个阶段，我们已经实现了 megatexture 的大部分目标，挑选一帧中能够看到的物体

用到的贴图，拼起来。这样我们就实时创建了一张贴图，这张贴图上面只有本帧需要绘制在屏幕上的贴图图素（当然肯定有一些冗余）。

那么如何绘制？ 因为我们用 pages 拼凑而成的这张显存贴图(physical page texture)，即便是相同物体用到的贴图，也都是一个个被切成小方块的 page 拼凑而成。page 与 page 之间根本不挨着。不同 mipmap 的 page 也都并列排在显存贴图里。根本不连续。用这唯一一张贴图去绘制场景所有物体。需要额外步骤。

纹理映射的原理是根据一个像素的贴图坐标(注意这里是已经被光栅化后的像素级贴图坐标了)，去贴图中采样图素（texel）。这个图素的颜色值，经过光照处理，就可以直接显示在屏幕像素上。

用 page 拼凑而成的显存贴图(physical page texture)正确显示场景中所有物体，需要对贴图坐标（物体贴图坐标是在 virtual texture 贴图空间中的，直接用这个贴图坐标去索引显存贴图（physical page texture），得不到正确结果）进行一下转换。使得显示在屏幕上物体的每个像素，精确的对应到显存贴图(physical page texture)上它应该对应的位置。

我们需要一次像素级别的贴图坐标转换。这个任务可以用 indirect texture sampler 也就是间接贴图采样来完成。即，先利用原始贴图坐标，对一张间接贴图进行采样，得到一些必要信息后，再利用这些信息对原始的贴图坐标进行转换以采集显存贴图。

这个间接贴图，其实就是页表贴图(page table texture)。

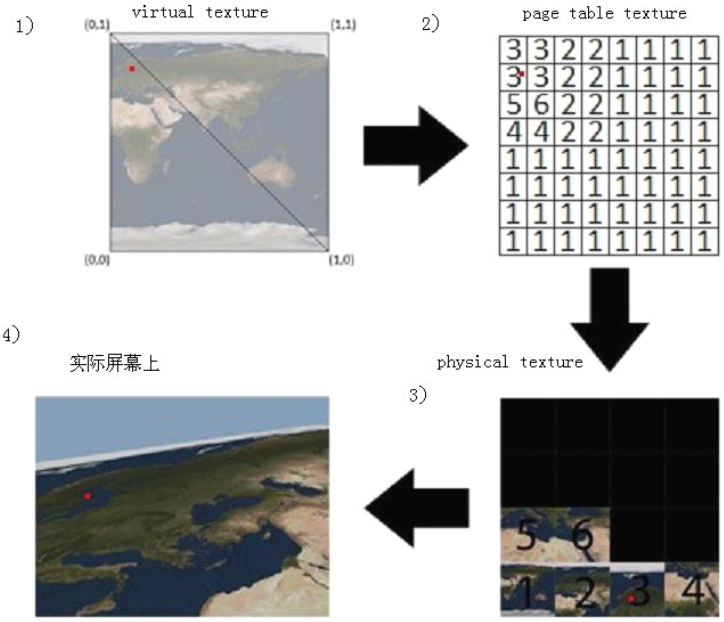
以下是过程简述：

1）用更新后的页表贴图（page table texture）与显存贴图(physical page texture)共同绘制场景所有物体

2）在 pixel shader 中进行一次贴图坐标间接寻址。从页表贴图（page table texture）中得到一个偏移值。用这个偏移值对原始贴图坐标（virtual texture 贴图坐标）进行处理，就变为最终需要采样显存贴图(physical page texture)的贴图坐标。

3）用处理过的贴图坐标采样显存贴图。最后场景中的物体就都按照实际需求绘制出来了。

如图：



这张图只是一个示意图。图 4 实际屏幕显示红点的位置，通过贴图坐标和光栅化我们可

以知道在 virtual texture 上采样的位置（图 1 中的红点）。用这个红点的贴图坐标，去采样页表贴图（page table texture，图 2），得到了当前活跃 page 在显存贴图中的位置换算关系。红点在图 2 落到了内容为“3”的图素里。利用这个换算关系，可以得到最终贴图坐标。在 图 3 中，正确的对显存贴图（physical page texture）进行采样。图 3 的红点就是需要正确采样的位置。最终得到图 4 像素的颜色值。

4，镜头快速转动与移动时

经过 3 个步骤，如果硬盘无限快速，而且显存贴图(physical page texture)的大小能够满足绘制本帧所有物体最大精度贴图。那么我们会看到一个无限大的世界，有着最高贴图精度，流畅地在眼前飘过。

事实上，没有这么理想的情况。 磁盘速度不可能那么快。显存贴图(physical page texture)有时候并不满足绘制当前镜头中所有物体最大精度贴图的容量。

这就是我们在实际《RAGE》游戏运行过程中看到的景象。一旦摇动一下镜头，所有物体的贴图都重新回到最模糊，然后慢慢变得精细。这个现象在 PS3 上尤其严重。PS3 的 IO 是严重瓶颈。理论上就无法保证足够量的精细贴图在一定时间内读入显存。在一个 光驱，硬盘，内存，显存的 4 级 cache 系统中，任何一个地方的瓶颈足以毁灭 megatexture 美好的初衷。这是为什么。

回顾 virtual texture 技术的 3 个步骤。如果磁盘是瓶颈，很多活跃 page 无法从磁盘调入显存。那么显存贴图(physical page texture)就无法在足够时间内拼凑出绘制当前场景所需要的全部贴图 page。 怎么办。 这个时候有很多办法可想。 一个最简单的办法是，没有贴图的景物，就绘制个马赛克贴图好了。。。

当然，商业产品不能这么做。 如果调入高精度贴图来不及，那么就先把最低精度的贴图调入进来。

如果最低精度都进不来，那么卡帧，让用户休息一下。。。

从这里看出，megatexture 受硬件制约依旧较大。只不过从明确的显存上限，降低到对 IO 速度的一定要求。

IO 速度对于 PC，可能越来越快。对于 XBOX360，卡马克爱不释手自然会量身定做。至于 PS3，那就算那些买正版的家伙们倒霉了。

5，一些问题

另外一种情况，当前帧的场景中物体太多了，所用到的贴图总量，超过了显存贴图(physical page texture)容量。

很多物体都没有贴图来绘制。自然只能空在那里。解决方案无非就是尽量合理的把场景中物体用到的贴图量控制一下。这里面要解决的工程问题无限多。要自研此技术做产品的人，只能自求多福了。

还有一个问题。在第一步用页表贴图(page table texture)绘制时，页表贴图是一张完整调入显卡显存的贴图。它是 virtual texture 的缩小版。根据 page 的大小来缩小。一个 page 相当于这个贴图的一个像素。但是当 virtual texture 中 page 数量太多，以至于页表贴图(page table

texture)的大小都超标了，这就是个很麻烦的事了。

我们假设一个 page 512x512 像素（page 大小的制定也有讲究，不细表）。如果想要在页表贴图(page table texture)保持在 4096x4096 大小，那么我们的一张 virtual texture 最大边长只能是 4096x512 像素。 也就是这张 virtual texture 最大也就 2097152 x 2097152 像素大小。一个像素算 4 字节。那么最大的一张图容量是 4096 G 。 难道我们与时俱进的玩家就按么容易满足 4T 的贴图容量么？随着祖国日新月异国富民强，人民群众的审美娱乐需求一日提高，早晚有一天。。。。。

如果真有这种情况，用多张页表贴图是一种自然的选择。。。

同时还存在其它问题。比如浮点贴图坐标的精度问题。显卡内部对贴图坐标浮点精度的处理和标准 IEEE 单精浮点标准是不一样的。另外，page table texture 是一张用来索引贴图坐标的贴图，这张贴图本身的内容就是贴图坐标，所以也需要用浮点格式。。。

在 virtual texture 很大的情况下，各种 buffer 各种贴图坐标浮点误差的处理，也令人头疼。

总结

基于 virtual texture 的 megatexture 技术。充斥着无数工程细节问题。各种多线程优化问题。各种 IO 优化，浮点误差，带宽瓶颈等问题。在 PC 上，ATI 显卡险些全军覆没也暴露了在兼容性方面的问题。其它还有技术与工具集生产流水线的集成问题等。

id software 在此领域推出成熟技术解决方案，令人钦佩。

megatexture 解决了静态纹理容量问题。但游戏的表现，并不完全取决于静态光影和贴图细致度。在 ID TECH 6 中，卡马克打算用 voxel rendering 把几何数据容量问题也一并解决掉，并且抛弃多边形光栅化渲染，带玩家走入另外一个细节复杂度无限高的虚拟现实世界。我们可以拭目以待。

2011－10

PS. H3D2 Engine

这几年公司开发了一款新的游戏引擎。采用了延迟渲染和前向渲染混合的管道。准备开发公司下一代的产品。由于产品在艺术表现方面的一些特殊需求，引擎工具主要在于营造逼真的舞台艺术效果。demo 中有 100 多盏实时光源。

延迟着色给美术带来的便利在于不再受到实时光源限制，可以用多钟光源任意营造想要的效果。次世代图像技术正在慢慢改变 3D 游戏图形开发的性质。使之成为一种更接近一种用技术创造艺术的过程。特效美术成为了舞台灯光设计师。角色美术也必须肩负起布料仿真的效果编辑。

技术只是一种手段，创造一些带有艺术感的新奇东西是件很有意思的事情。
截图视频等信息在：
http://www.cnblogs.com/puzzy3d/archive/2011/05/04/2036327.html
视频
http://h3d.com.cn/downloads/h3d2_engine-run.mp4