Dezeming Family

PBRT 系列 19-专业知识理论与代码实战-运动模糊与实例化

Copyright © 2022-07-22 Dezeming Family

Copying prohibited

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, without the prior written permission of the publisher.

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列电子书，可以从我们的网站 *[https://dezeming.top/]* 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

0.1 本文前言

由于我们一直没有实现场景 Parse 系统，所以我们需要在程序中直接定义加载的每个模型和场景，久而久之，程序就变得非常庞大和杂乱，因此，在本系列中，我们将生成模型、材质和光源的一些函数分别放在了 ModelSet.h、MaterialSet.h 和 LightSet.h 三个头文件里。

本文主要内容是运动模糊和实例化，它们会让系统变得更加复杂。虽然有不少需要移植的内容，但它们不会影响到后面的着色计算，仅仅会影响到光线求交；同样，实例化也是只会需要改变一些光线求交和包围盒方面的代码，不会影响其他内容。本书内容紧跟 PBRT 系列 18，本书的代码也是在 PBRT 系列 18 的基础之上来构建的。

我们的系列书全都是免费公开的，但由于服务器实在是昂贵（一开始我们买的是联机打游戏用的服务器，自从光子映射作为第一本电子书被放在网站上以后，DezemingFamily 才正式开始，后来我们也不想再花费精力换服务器了），所以希望觉得本系列有帮助的读者们能够给予一定的支持。本系列全套售价为 120 元——当然，哪怕是赞助 5 块钱，我们都会很开心——免费公开的内容如果大家愿意支持，说明我们的系列书得到了认可，我们自然是非常开心的。

1. 运动模糊

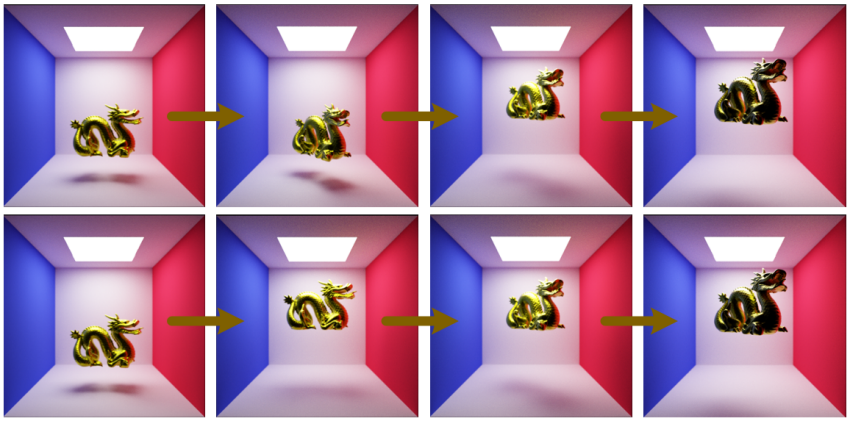
本章讲解 *PBRT* 中运动模糊的实现过程。

1.1 运动模糊的基本概念与实现方案

PBRT 第一版是不支持运动模糊的，从第二版开始才逐渐支持了运动模糊。运动模糊就是模拟相机在有限时间中捕获的运动物体带来的模糊效果，在工业电影中应用非常广泛（可以说，工业渲染器中任何模型都需要设定其运动状态），运动模糊有两点我们需要注意，一是对某个时刻物体的位置做插值，二是考虑如何构建相应的加速包围盒。

为了支持运动模糊，我们需要知道在一个时间段内物体在各个时间点的位置和状态，比如我们目前有一个模型，它在一个时间段内逆时针沿着 y 轴旋转了 90 度，那我们如何知道它是逆时针转 90 度还是顺时针转了 270 度呢？这两种理解方式渲染出来的结果是不太一样的，一个看起来转得更快了。所以，我们一般会定义较小的时间片段，比如 1/23 秒内，那么这样的好处是物体在这么段的时间片内运动范围相对较小，那么该时间段内的各个时间节点插值结果就会更准确。

那么下一个问题就是怎么插值：我们有一个时间段内的初始的矩阵和末尾的矩阵，那么，怎么得到中间的矩阵呢？



在极短的时间段内，我们不需要考虑模型是先旋转再平移还是先平移再旋转，我们假设模型是“一边平移一边旋转”的，我们对这个过程进行插值。

PBRT 的方式是进行矩阵分解，将一个变换矩阵 **M** 分解为三个矩阵 **M** = **SRT**，其中，**S** 表示尺度缩放，**R** 表示旋转，**T** 表示平移。然后分别对这三个矩阵插值，插值完以后再合并为当前时间点的变换矩阵。

平移和缩放的插值非常简单，但旋转的插值的就比较困难了，所幸借助四元数工具可以很好地进行插值。我们这里不讲解四元数和四元数插值的原理，有兴趣可以参考 DezemingFamily 的《四元数在三维旋转变换的应用》。

1.2 AnimatedTransform 类

本节介绍 AnimatedTransform 类的基本功能。

该类的构造函数包括时间片段内的起始的变换和末尾的变换，以及起始时间和末尾时间。

Decompose() 成员函数就是用来做矩阵分解的，Interpolate() 函数给出当前时间点，得到插值的变换。

运动变换类可以作用到 Ray 类对象、RayDifferential 类对象、Point3f 类对象以及 Vector3f 类对象。

actuallyAnimated 是一个 bool 类型的变换，表示初始变换是否等于末尾的变换。

MotionBounds() 和 BoundPointMotion() 都与绑定移动的包围盒有关，DerivativeTerm 结构与包围盒绑定有关，这些我们下一节再介绍。

Decompose() 函数在书 [1] 中有非常详细的解释，这里不再赘述。

1.3 绑定移动的包围盒

给定通过运动变换类来变换的 Bounds3f，能够计算其在运动时间段内所有运动的边界框是很有用的。例如，如果我们可以绑定运动的几何基元，那么我们可以将光线与该边界框相交，以确定光线是否可能与对象相交，然后再将基元的包围盒插值到光线所在的时间以检查相交。

有两种简单的情况：首先，如果关键帧矩阵相等，那么我们可以只任意应用开始变换来计算整个边界包围盒。其次，如果变换仅包括缩放和/或平移，则包含了开始时间和结束时间的边界框位置的边界框将限制其所有运动（这句话比较拗口，其实就是开始时间和结束时间的边界框可以直接通过 Union 合并为最后的边界框）。

包含了旋转的边界框就会麻烦一点，因为一个轴对齐的包围盒想要包括一个轴对齐的立方体，则取立方体的最大和最小坐标值就可以了。但如果立方体稍微一旋转，再求包围盒就会麻烦很多。

我们的三角形面片都会在一开始就作用于变换矩阵，然后给其安排包围盒，如果是动态变换，则情况会有所不同，我们在后面会再介绍。虽然去包围包含运动旋转变换的物体比较麻烦，但仅仅为了移植的话其实并不需要了解太多，大家有兴趣可以在 [1] 中了解这个过程。

我们当前已经了解了运动变换包围盒需要关注的问题，我们可以动手将 AnimatedTransform 类和 Quaternion 类移植到我们的系统里。需要移植和注意的地方我进行了总结：

内容一：在工程总头文件中声明：

```
struct Quaternion;
class AnimatedTransform;
```

内容二：在 Transfrom 中定义友元：

```
  friend class AnimatedTransform;
  friend struct Quaternion;
```

内容三：移植 quaternion.h 和 quaternion.cpp 两个文件中的全部内容到我们自己的系统中。

内容四：移植 AnimatedTransform 类及相关功能到我们的系统中。

编译通过即可。

1.4 动态变换与移动的物体

有哪些对象会使用到 AnimatedTransform 类？我想大家肯定是觉得头大，好像我们需要把代码里全部的与变换有关的内容都找出来然后改为 AnimatedTransform，然而，真的需要这样吗？好在借助电脑我们可以很容易地进行搜索，我们将 PBRT 中全部与 AnimatedTransform 有关的内容都列出来：

- 全部 Camera 类及其子类。
- TransformedPrimitive 类。

没想到竟然只需要关注这两个部分！当我发现仅有这两个类会直接使用到 AnimatedTransform 类时，我瞬间觉得茅塞顿开——运动模糊相关功能其实只占据了相当一小部分代码！

我们在讲解动态变换之前，先把 TransformedPrimitive 类给移植完毕，并且保证能够编译通过。然后把每个相机类中的 Transform CameraToWorld 都改为 AnimatedTransform CameraToWorld，然后在构造函数中加入这两个变量：

```
  float shutterOpen
  float shutterClose
```

注意相机的 GenerateRay() 函数和 GenerateRayDifferential() 函数里：

```
  ray->time = Lerp(sample.time, shutterOpen, shutterClose);
```

Ray 上携带有时间信息，当 Ray 与包围盒求交时，存在 TransformedPrimitive 基元的包围盒就会进行插值，得到当前状态的包围盒，然后再进行求交测试，这个过程还是比较耗费时间的。

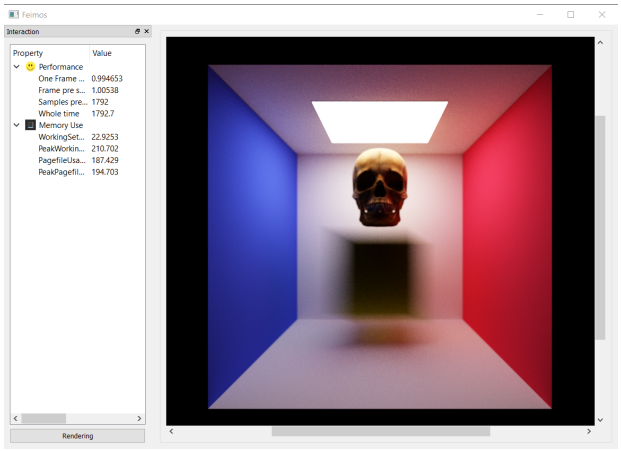
制作动态物体的变换矩阵：

```
Transform tri_Object2WorldStart;
Transform tri_Object2WorldEnd;
Feimos::AnimatedTransform animatedTrans(&tri_Object2WorldStart, 0.0f,
   &tri_Object2WorldEnd, 1.0f);
```

在 getMovingDragon() 函数中：

```
for (int i = 0; i < plyi.nTriangles; ++i) {
  std::shared_ptr<Primitive> geoPrim = std::make_shared<Feimos::
      GeometricPrimitive>(tris[i], material, nullptr,
      mediumInterface);
  std::shared_ptr<TransformedPrimitive> aniPrim = std::make_shared<
      TransformedPrimitive>(geoPrim, animatedTrans);
  prims.push_back(aniPrim);
}
```

貌似渲染包含有运动模糊的物体耗费的时间太多了，所以首先，我只让一个立方体盒子做为运动物体。渲染得到的结果如下：



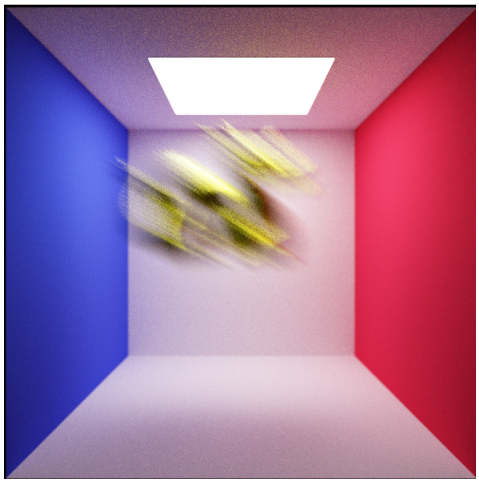
当然，在学完了下面的实例化过程后，读者可能会想到，为什么不给整个模型建立一个统一的运动接口呢？也就是说，给模型自己生成 BVH 树，然后把这个树的根节点赋值给 TransformedPrimitive 对象：

```
// 把每个模型作为一个整体
std::vector<std::shared_ptr<Feimos::Primitive>> primsObj;
for (int i = 0; i < plyi.nTriangles; ++i) {
  std::shared_ptr<Primitive> geoPrim = std::make_shared<Feimos::
      GeometricPrimitive>(tris[i], material, nullptr, mediumInterface)
      ;
  primsObj.push_back(geoPrim);
}
// 为模型建立自身的加速结构
std::shared_ptr<Feimos::Primitive> aggregate = std::make_unique<Feimos
    ::BVHAccel>(primsObj, 1);
// 为模型构建可变动基元
std::shared_ptr<TransformedPrimitive> aniPrim = std::make_shared<
    TransformedPrimitive>(aggregate, animatedTrans);
```

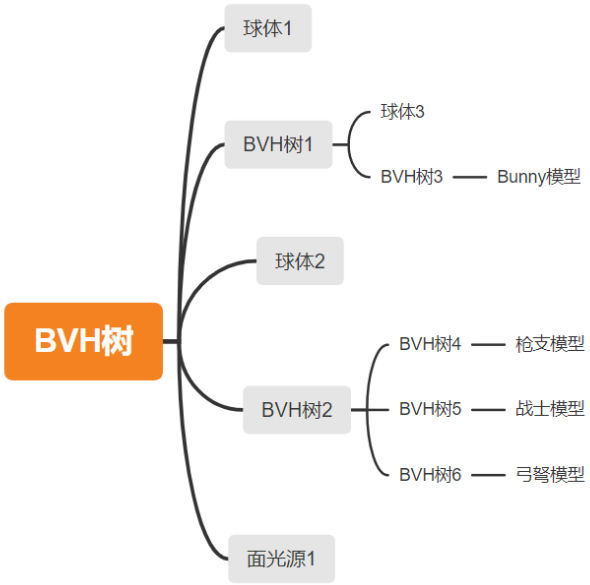
```
11 // 填充到场景基元数组中
12 prims.push_back(aniPrim);
```

大家可以把这段代码自行更改到 getMovingDragon() 函数中。

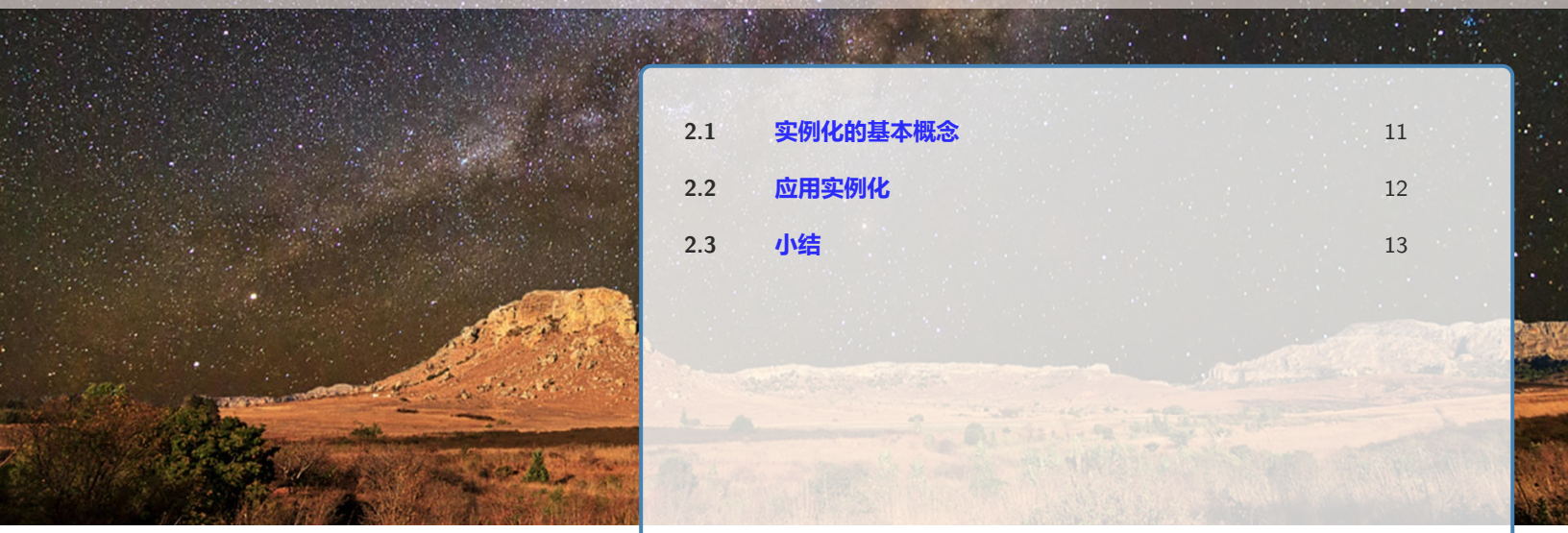
这样，渲染的速度就会非常快了（比前面的方式快了二十倍）：



将整个模型作为一个“大”基元，为这个基元的内部构建加速结构，是一种非常自然的方式，其实我们在一开始学完加速结构的时候就可以这么做了，但由于本人的一些疏忽（而且，不使用实例化或者运动模糊，构建每个模型的基元加速器带来的性能提升并不会比整体构建加速器高多少），而且，对于复杂场景，或许我们需要尝试更好地基于空间来安排 BVH 树的结构，这样会增加一点复杂性：



2. 实例化



本章讲解 *PBRT* 中实例化的实现过程。

2.1 实例化的基本概念

所谓实例化 (Instancing)，就是同时创建大量相同的物体，但是这些物体有着不同的位置（做了不同的变换），这个过程就叫做实例化。

在 OpenGL 中，实例化意味着避免通过 CPU 大量调用 glDrawArrays 这类绘制函数，因为 CPU 与 OpenGL 通信消耗的时间远大于 GPU 处理顶点和着色的时间。

在 PBRT 这种离散渲染器中，实例化需要其他的处理手法。我们看到 pbrtObjectInstance() 函数，该函数会定义实例化的内容。

如果实例化对象包含多个基元，则需要提前将这个实例化对象生成加速结构。我们直接修改模型类 RattlerLoad() 函数：

```
1  std::vector<std::shared_ptr<Feimos::Primitive>> primsObj;
2  for(int j = 0; j < trimesh->nTriangles; ++j){
3    primsObj.push_back(.....);
4  }
5  // 在每个模型内部构件好加速结构
6  std::shared_ptr<Feimos::Aggregate> aggregate = std::make_unique<
       Feimos::BVHAccel>(primsObj, 1);
7  prims.push_back(aggregate);
```

这样相当于给每个模型都生成一个包围盒。

为什么实例化对象需要用到 AnimatedTransform 类来管理呢？这是因为比如以三角形为例，它的位置早就是我们已经提前定义好的（同理，其他物体的包围盒也都是构建好物体位置以后就

确定了的），但我们希望大量重复使用该物体，则 AnimatedTransform 类相当于给这个已经固定好的物体再施加一次变换，使其包围盒变到其他位置。

2.2 应用实例化

我们给模型生成函数再添加两个参数：

```
std::vector<AnimatedTransform>instancingTransform
bool isInstancing
```

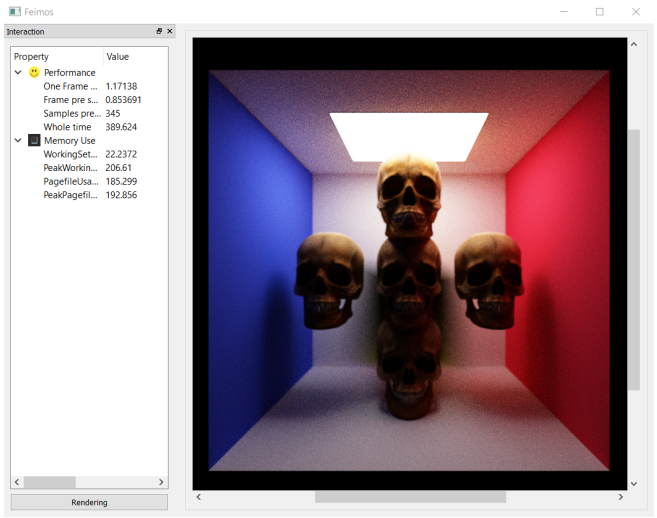
两个参数一个是实例化变换的数组，另一个是是否使用实例化。

构造实例化的代码如下：

```
// 模型构建面片
......
// 在每个模型内部构件好加速结构
std::shared_ptr<Feimos::Primitive> aggregate = std::make_unique<Feimos
    ::BVHAccel>(primsObj, 1);
// 构造实例化
if (!isInstancing || instancingTransform.size() == 0) {
  prims.push_back(aggregate);
}
else {
  for (int i = 0; i < instancingTransform.size(); i++) {
    std::shared_ptr<Feimos::Primitive> pri = std::make_shared<
        TransformedPrimitive>(aggregate, instancingTransform[i]);
    prims.push_back(pri);
  }
}
```

由此可见，实例化并没有额外增加什么基础设施代码，只使用我们新增加的 TransformedPrimitive 基元类就能够实现。

构造多个实例化物体渲染结果如下：



可以看到并没有增加太多的渲染时间。

2.3 小结

本文于 2022 年 7 月 22 日完成，总共耗费一天时间，这也是因为我将矩阵分解、四元数插值和运动包围盒生成的详细细节省略，才能够一天就完成本文的写作。当然，因为 PBRT 书中对这些知识点的讲解非常详细了，所以我没有再赘述的必要（这些知识点跟物理材质或者渲染概率方法不同，不掌握具体细节对我们的后续深入研究并不会造成什么阻碍，所以就此省略）。

目前我还没有开始着手写物理材质一书（PBRT 系列 16），因为系列 16 并不会增加什么代码，而且我打算把系列 16 和系列 21（次表面散射）用两周时间一口气写完。下一本书就将是一场硬仗——PBRT 最重要的组成部分——物理材质；同时，我会再次重新修订一下 PBRT 系列 20（渲染概率与采样）。

感谢大家能坚持阅读到这里！

参考文献

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] https://learnopengl.com/