# 计算机图形学与角色群组仿真

# COMPUTER GRAPHICS AND CROWD SIMULATION

陈红倩 著

机械工业出版社
CHINA MACHINE PRESS

# 计算机图形学与角色群组仿真

陈红倩　著

机 械 工 业 出 版 社

本书以群组仿真技术的实现过程为主线，综合群组仿真涉及的几大模块，针对模型变形技术、动作驱动技术、快速绘制技术以及硬件加速技术进行探讨和阐述。书中包含了大量的实例介绍和代码示例，具有一定的视角广度和技术深度。

本书可以作为运动仿真和角色群组仿真方面的参考书，还可以作为计算机图形学领域技术人员的提高性参考书，对于图形硬件编程人员也具有参考与实用意义。

**图书在版编目（CIP）数据**

计算机图形学与角色群组仿真/陈红倩著. —北京：机械工业出版社，2011.4

ISBN 978-7-111-33883-3

Ⅰ.①计… Ⅱ.①陈… Ⅲ.①计算机图形学②计算机仿真 Ⅳ.①TP391.41②TP391.9

中国版本图书馆 CIP 数据核字（2011）第 051183 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）

策划编辑：牛新国 吕 潇 责任编辑：吕 潇

版式设计：霍永明 责任校对：刘怡丹

封面设计：路恩中 责任印制：乔 宇

三河市国英印务有限公司印刷

2011 年 5 月第 1 版第 1 次印刷

169mm × 239mm · 10.75 印张 · 214 千字

0001—2500册

标准书号：ISBN 978-7-111-33883-3

定价：39.80元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

社 服 务 中 心：（010）88361066

销 售 一 部：（010）68326294

销 售 二 部：（010）88379649

读者购书热线：（010）88379203

网络服务

门户网：http://www.cmpbook.com

教材网：http://www.cmpedu.com

**封面无防伪标均为盗版**

# 前　言

随着计算机图形学技术的发展，3D 动画技术、虚拟现实技术以及数字电影制作技术出现了空前的进步，令各领域的专家们都刮目相看，从而进一步激发出了大量的新应用、新热点。在军事演习、人群行为模拟、大型舞台节目设计、电影战争场面制作等应用中，由于涉及大量的运动人物或其他个体，因此达到理想的最终效果需要非常长时间的编排和训练，花费大量的人力物力。虚拟现实的发展为这类场景的设计提供了一个完善的仿真环境，通过设置个体的行为模式可以迅速地实现整体的场景效果，反之，还可以根据整体场景效果，计算出个体的行为模式，从而指导实际训练。

随着奥运会开幕式、阅兵式、国庆晚会等大型表演受到全国人民的喜爱，科研工作者对角色群组仿真技术也越来越关注。如何使用计算机仿真的方法，对实际演习、演练、编排起到更好的辅助与加速作用，并提高实际展现的效果水平，是一个热门的研究课题。

角色群组仿真技术是一种虚拟现实领域的技术，它通过计算机建模手段将最终场景中的实体进行建模，并通过骨骼、蒙皮动画等手段驱动场景中的实体，辅以大规模绘制技术，实现大规模运动角色的快速仿真。如在电影《指环王 3：王者无敌》中 6 万人的战争场面，就是一个大规模群组运动仿真的典型应用。同时，角色群组仿真技术还包含场景编排的逆向过程，即将整体的场景效果，通过分解计算得到场景中每个个体的动作，这对于实现导演意图的快速编排具有极大的帮助。

在计算机图形学领域中，角色动作驱动、角色变形与角色快速绘制方法是角色仿真过程中的三个必要过程，也是大规模角色群组仿真的基础。针对这三方面的研究，近几年来，国内、国际的期刊均有相当多的论文发表。

本书以角色群组仿真技术的实现过程为主线，针对其所涉及的模型变形技术、模型驱动技术、快速绘制技术以及硬件加速技术进行探讨和阐述。在每章中，首先介绍这类技术的当前概况、技术原理以及主要实现方法，然后介绍本书作者所提出的新方法、新技术，并针对该技术的优越性与之前的方法进行对比，最后使用该方法实现应用实例，并展示实例结果。在每章的内容中，都引用了大量的参考文献，针对当前的技术概况进行了深入的分析与归纳总结，包含了大量的实例介绍和代码示例，具有一定的视角广度和技术深度。

本书针对角色群组仿真过程中几大模块的技术难点、重点进行了详尽的阐

述，对技术的实现过程和应用实例进行了讲述，并概述了所涉及的图形学基础，以及图形硬件加速方法。本书的内容主要包含如下几部分：

（1）计算机图形学与群组仿真概述。该部分主要介绍群组仿真技术的应用背景、技术概要，群组仿真技术的几大模块的相互关系等内容。

（2）角色变形技术。变形技术是所有运动仿真技术的基础，所有运动的对象都将通过变形实现其运动。在本书中将分别介绍和研究针对2D图像角色和3D模型角色的运动变形技术。

（3）运动驱动技术。使角色真实地运动起来，需要准确且合理的运动数据。在本书中将分别就运动数据的获取与转换、迁移、应用等方面进行介绍和探讨性研究。

（4）快速绘制技术。随着群组仿真中运动个体的数量越来越多，绘制成为群组仿真的一个瓶颈。在本书中，将就当前的快速绘制技术进行介绍，并介绍本书作者在快速绘制方面所取得的一些研究成果。

（5）硬件加速技术。提高仿真与绘制速度，离不开计算机资源的有效利用，计算机图形硬件的发展，使得图形硬件成为一个重要的计算单元，目前最新的操作系统和大软件，均已公布将利用GPU（Graphics Process Unit，图形处理单元）提高运行效率的计划。本篇将介绍使用图形硬件进行开发的原理和步骤，以及在群组仿真中的具体实现方法。

本书可以作为运动角色仿真或角色群组仿真方面的参考书，还可以作为计算机图形学领域的技术研究人员的提高性参考书，对于图形硬件编程也具有一定的参考与实用意义。

本书的基础是作者本人博士研究期间所取得的成果，书中所附带的源代码均曾由作者本人调试通过并用于实验。在此感谢导师战守义教授、李凤霞教授对本人的精心指导，感谢本书作者原博士研究单位北京理工大学计算机学院的培养，感谢曾为本书作出贡献的同学、同事、朋友。本书的出版得到了北京工商大学各级领导的关心、支持和帮助，并得到了北京工商大学“北京市属市管高校人才强校计划”项目的资助，书中参考了国内外许多专家、学者的论著，在此一并致以衷心的感谢。

由于本书所涉及的课题内容较新，一些理论方法和技术还在继续研究之中，加之作者水平有限，书中错漏之处在所难免，欢迎读者批评指正。另外，由于本书涉及大量图形学的相关知识，或有个别引用遗漏之处，在此一并致歉。

作　者

2011年4月

# 目　录

# 第 1 章　群组仿真相关技术概论

科学技术的发展极大地扩展了人们的视野，也极大地开拓了人们的知识面，同时为提高人们对信息的理解能力，增强人的知识接收能力作出了杰出贡献。据科学研究表明，人所获取的信息中 70% 来自于眼睛，也就是说，大量的信息来源于视觉所感知的图形图像。自从计算机开始进入人们的视野以来，从指示灯到文字的打印输出，再到现在的多媒体技术，图形图像占据着计算机时代的大部分输出形式。

随着计算机处理能力的提高，人们已经不再满足于文字的交流，还希望通过人们的视觉、听觉、触觉，乃至形体、手势等参与到信息处理的环境中。这种信息的处理，已经不仅仅是针对一维的文字信息，而是针对多维的图形图像信息了。在这个过程中，计算机图形图像作为人机交互界面的主要技术支撑部分，发挥了巨大的作用，从而也成为了一个庞大的研究领域。

## 1.1　计算机图形学

“计算机图形学”是一门研究图形图像处理的学科，在英文中被称为 Graphics，在国内它是计算机专业学生的一门必修课，针对该门课程的教材和参考书比比皆是，但笔者经过比对发现，各大专院校针对这门课的教学计划不尽相同，不同的图形学的教材所涉及的内容也差之甚远。其实，在图形学领域一直对计算机图形学的研究范畴有较大的争议，尤其是近些年来，随着三维图形学的发展，对于计算机图形学的涵盖内容存在较大的出入。

最初的图形学研究主要集中于如何在计算机屏幕上快速地显示文字、图像等内容（如快速地绘制出一条直线、快速地填充一个三角形区域等），以及如何充分利用有限的帧缓冲区——显示内存（显存）进行复杂的图像输出。最初的图形学研究集中于这些领域主要有两个原因，一个原因是受当时的计算机硬件条件限制，在计算机开始普及的初期，各硬件的价格都非常高昂，为降低计算机价格，不得不配置较少的内存、显存等，这对快速的内容显示提出了很高的要求；另一个原因是由于在当时的计算机显示需求中仅限于将文字内容、操作命令进行图像化，以利于更方便的操作。

摩尔定律指出集成电路上可容纳的晶体管数目，约每隔 18 个月便会增加 1

倍[⊖]，性能也将提升 1 倍，这一定律揭示了信息技术进步的速度。随着计算机硬件的发展，使得 PC 获得了越来越强的处理能力，人们在追求更快的处理速度的同时，也追求更佳的用户界面以及更好的使用体验。如在操作系统方面，从 Windows 2000、Windows XP，到 Windows Vista、Windows 7，操作系统的界面由标准界面过渡到 Windows XP 的“炫彩”界面，到 Windows Vista 和 Windows 7 的 Aero 玻璃效果，都吸引了大量用户群的喜爱和追捧。这些用户体验的提高，极大地促进了计算机图形学的发展，同时也引发了大量新技术和新领域的应用。

计算机硬件的发展，使得计算机图形学技术得以巨大的发展。以三维建模技术、纹理技术、混合技术、绘制技术等为代表的新技术成为人们研究的重点，而针对于具体应用所衍生出的变形技术、骨骼技术、蒙皮动画等逐渐成为计算机图形学所研究的主题。计算机图形学的应用领域，也从之前的计算机平面绘图发展到更加广泛的领域，如数据可视化、计算机仿真、电影动漫、游戏技术、装配制造、编排训练等方面。

进入 21 世纪后，可编程图形处理单元（Generic Programming Unit，GPU）的出现，使得计算机图形学技术的发展登上了一个更高的台阶，大量的应用又开始往并行性和硬件运行等方面发展。从以往追求更佳的显示效果，开始发展为效果和性能并重，倾向于实时性大规模应用的发展之路。

在计算机图形学业界内，为区分早期的计算机图形学和近期图形学的研究内容不同范畴，将最初期的计算机图形学称为二维图形学，也叫做光栅图形学，将近期的以计算机三维建模为基础的图形学技术称为三维图形学，有时也称为真实感图形学。

### 1.1.1 计算机二维图形学

在计算机二维（2 Dimension，2D）图形学中，由于图形学算法的大量应用限制于计算机硬件的计算能力，研究者们在这方面花费了大量的精力，进行了深入的研究，也出现了很多著名的算法。计算机二维图形学中的一些关键技术包括几何变换、裁剪算法、扫描变换算法以及隐藏面隐藏线消除等算法。这些二维图形学中的算法，同样可以应用在三维（3D）图形学中，只是它们已经不再是研究的重点内容。在各位读者学习的过程中，也会涉及这些算法与技术。这些算法在某些方面也与本书的主题——计算机群组仿真技术相关，是几乎所有图形学技术及应用的基础。

---

⊖ 据查阅，“每 18 个月便会增加 1 倍”是后人修改的修正版本，摩尔最初的说法是“每年翻 1 番”。

**1. 图形学中的几何变换**

在计算机图形学中，无论是二维图形学还是三维图形学，都离不开几何变换，以三维图形学的观点看来，二维变换是三维变换的特例。在几何变换算法中，变换是通过矩阵对向量的操作实现的。每一次几何变换都可以对应为一个变换矩阵，每一个顶点的坐标使用一个向量来表示，顶点的位置改变相当于将其坐标与变换矩阵相乘，从而得到一个新的坐标向量。当一个物体的所有顶点位置发生了同样的位置改变，而顶点之间的相对位置不变，那么就相当于针对该物体完成了位置改变。当物体的各个顶点相对位置发生了不同的改变，则该物体发生了变形操作。

**2. 裁剪算法**

裁剪算法是计算机图形学中的一个重要部分，对于计算机图形学的执行速度有重大影响。裁剪是针对一个场景中物体可见性进行判断的算法，对于一个计算机所显示的场景来说，在逻辑上可能有上千上万件物体，但在计算机屏幕上，能同时被看到的可能只有几件，那么，对于本次计算机绘制来说，看不到的这些物体，就可以排除在当前的显示计算过程之外。

对于一个二维平面图像来说，看不到的部分包括屏幕外上下左右四个部分。由于图形系统中的每一个图形基本元素，都需要进行裁剪判断工作，因此，能否快速地判断物体是否位于窗口内，直接影响整个图形系统的效率。

裁剪算法有很多，效率的高低常常与所渲染的场景情况有关，因此需要根据实际情况选择裁剪算法。Sutherland-Cohen 算法是一种针对直线的快速裁剪算法，该算法通过计算线段的两个端点与屏幕的位置关系，获取直线是否位于屏幕内。梁友栋-Barsky 算法是另一种经典的图形裁剪算法，它不再考虑线段与屏幕边界的位置关系，而是直接将线段所在的直线与四条屏幕边界线方程进行求交，根据交点排序而决定裁剪结果。

**3. 多边形的扫描变换**

在计算机图形学的内部表示中，物体使用点、线、面、体表示，如三角形的图形只需要存储其三个顶点的位置即可。而要在计算机屏幕上显示，需要转换成像素表示，也称为点阵表示。顶点表示是用多边形的顶点序列来表示多边形，该方法几何意义强，占有内存少，但它不利于在屏幕显示时进行像素着色。点阵表示是用位于多边形内的像素的集合来表示多边形，该方法不包含多边形的几何信息，但便于使用帧缓冲器存储图像，并在屏幕上显示。

图形由顶点表示转换为点阵表示的过程称为光栅化。以多边形为例，图形的光栅化，即求出多边形内部的各个像素，并在帧缓冲器的对应元素上设置相应的灰度和颜色。这个过程也称为多边形的扫描转换，其转换算法主要有逐点判断法、扫描线算法、边缘填充算法、边界标志算法等。

**4. 隐藏面和隐藏线的消除**

隐藏面和隐藏线的消除是计算机图形学中的另一个基本问题，讨论关于图形在二维屏幕上显示时，如何消除二义性的问题，消除的结果是使得图形能够反映不透光物体中被遮挡的部分。

消除算法的主要原理就是对曲面按 $Z$ 值的递减顺序（即由前到后）进行排序，然后计算各曲线或曲面间的交线，针对交线之间的某一个区域内，选择位于前面的曲线或曲面作为该区域的最终颜色。Z 缓冲器算法是最简单的隐藏面消除算法之一，用一个 Z 缓冲器存储着一个屏幕（$X$，$Y$）位置上的最大 $Z$ 值，对于每一个显示对象，与相应的 Z 缓冲器内的值进行比较，如果更靠近观察者则修改颜色显示，并同时修改 Z 缓冲器的值，直至所有对象绘制完成。

对于隐藏面和隐藏线的消除，进一步的算法改进有扫描线算法、区域子分算法、曲线扫描线算法以及区间扫描线算法。

### 1.1.2 计算机三维图形学

随着图形学硬件的发展，人们对于视觉效果的追求也越来越高，技术人员的关注点也已不再是如何提高裁剪算法和扫描算法的速度，而是如何获得更优质的显示效果。

在计算机图形显示设备上生成一幅高真实感的场景图像，一般需要经过场景描述（即建模过程）、坐标变换、颜色与纹理映射、绘制与显示等几个过程。在这几个过程中，坐标变换过程是最难以理解的部分，在新手学习过程中也是最容易出错的部分。如果说其他部分关系到显示效果的优劣，那么坐标变换过程则是场景能否被正确显示、能否被看到的关键。

三维坐标变换的过程包括四个步骤，它们依次是世界变换、视图变换、投影变换、视窗变换。其中世界变换实现主要负责场景中各个物体之间的位置关系，将物体坐标由其自身为基准的坐标，变换到整个场景统一的世界坐标。视图变换将物体坐标由世界坐标变换为以观察者为基准的视图坐标系中。投影变换将观察者看到的世界从三维世界中投影到二维平面世界中。投影变换分为两种，一种是正射投影，另一种是透视投影。正射投影多用于建筑蓝图或工业模型设计，在投影过程中，同样大小的物体投影后占据屏幕大小也相同；透视投影多用于动画模拟和真实感图形图像，在透视投影中，同样大小的物体当离观察者距离较远时所占屏幕面积较小。视窗变换根据最终显示窗口的大小，将图形学中的逻辑坐标变换为操作系统中的像素坐标。

随着计算机硬件的发展，尤其是图形硬件的发展，计算机图形学中的经典问题，如裁剪问题、多边形转换问题、消隐问题都已不再是图形学的瓶颈，取而代之的是建模问题、控制问题、纹理贴图、光照计算、硬件加速等问题，而在近几

年的图形技术发展过程中，很多问题已经由研究者们提供了大量的解决方案。目前，在三维图形学中仍然关注的研究重点有如下这几个部分：

**1. 三维模型建模技术**

三维图形学研究的第一个方面是如何快速地模拟真实世界或建立虚拟世界中的场景，因此需要在整个场景中观察什么东西，以及场景中展示的物体如何获得是一个关键问题。随着计算机处理能力的提高，人们对可视化效果的要求越来越高，所表示的模型规模越来越大，模型的细节程度也越来越详细，详细庞大的模型使得建模过程越来越复杂。研究快速建模方法，寻找更优质的模型表示方法以及使用更少的数据准确地表示模型，是三维图形学所研究的一个问题。

在三维图形学中，制作虚拟物品一般有下面三个途径：一是通过编程方法直接生成，二是使用商品化的软件辅助制作，三是通过摄像机或三维扫描仪进行拍摄。在技术实现上，建模基本可分为两步：第一步称为几何建模，主要包括用多边形或三角形拼构成对象的立体外形；第二步称为物理建模，主要包括对几何建模的结果进行纹理、颜色、光照等处理。用几何建模建立模型主要有三种方法：线框模型（也称为多边形建模）、曲面模型和实体模型。

对于上述所述的三种建模方法，多边形建模是被广泛使用的方法，在屏幕上看到的几何图形由许多互相连接的小三角形组成，这些三角形被称为“面片”。每个面片有不同的尺寸和方向，同一模型的面片数越多，则模型的细节就越细腻。

曲面建模是通过曲线（通常是 Bezier 曲线）表示表面区域边界而实现的模型定义方式，边界线之间的区域是平滑过渡的，曲面建模技术可以使用很少的细节表示出很光滑的形状，同时这种表示形式可以实现快速参数化变形。

实体造型技术是基于计算机辅助设计和制造发展起来的建模技术，能处理类似于打孔等复杂的模型表示，并可以实现自动消隐计算、侧影轮廓线等模型特征求解，对于模型的重心、体积等也可以很快求解。实体造型技术包括 CSG 体素构造表示、实体边界表示、八叉树表示、基于特征的实体造型等方法。

**2. 模型变形及匹配技术**

三维图形学需要研究的第二个主要方面是模型变形。图形学中所展现出来的模型，只有少部分是通过原始建模过程建出来的，更多的模型是通过变形技术实现的。随着应用领域的扩展，变形技术变得更加重要，而超大模型的变形和大规模模型的快速变形技术成为了一个研究难题。本书中所研究的很大一部分内容，就是进行模型变形方面的研究，相关章节会给出更为详细的模型变形方面的技术现状及前沿信息。

**3. 三维场景快速绘制技术**

三维图形学研究的第三个方面是快速绘制技术。随着图形学的发展，现在的

仿真应用的技术中，对模型的精细程度以及仿真规模，都提出了更高的要求对于仿真应用中，实时性是一个需要着重考虑的因素，因此研究超大模型或大规模仿真应用的快速绘制技术，是三维图形学中研究的一个课题。

大规模场景的实时绘制问题是虚拟场景实时生成和可视化的关键问题，也是实现实时交互性的前提。计算机三维图形学中，加快场景渲染速度往往有两种途径：一种是提高图形系统的性能，另一种就是减少场景中所需渲染的面片数。减少场景所需渲染面片数的三个重要途径：场景面片剔除、LOD 简化和基于图像的渲染。

对场景面片进行剔除是减少计算量十分明显的一种加速方法，在图形运行库（如 OpenGL 和 DirectX）中，每个面片都根据顶点顺序分为正面和背面。在渲染过程中可以指定渲染其中一面，这对于闭合物体，将减少一半的纹理和光照计算量。除图形系统运行库所内置的背面剔除算法之外，在系统渲染中所常用的剔除方法还有：视景剔除、遮挡剔除、误差合并。

视景剔除是指对场景中的物体，根据当前视点的位置和视域范围，计算物体的可视性，对当前视点下不可见的物体进行剔除。视景剔除算法一般来源于二维图形学中的裁剪算法。遮挡剔除是指对经过视景剔除保留下来的物体，根据当前的视点与面片的位置关系，通过计算面片的相互遮挡关系，剔除被遮挡的面片。误差合并是指根据面片对最终屏幕图像的贡献率，当贡献率小于既定的阈值时，对该面片进行剔除，并将其两侧的面片合并以填补这个形成的孔洞。

图形渲染过程中，使用 LOD 方法进行加速已经成为了一个标准式方法，大规模场景绘制中的 LOD 方法，基本经历了离散层次细节技术到连续层次细节技术的历程。LOD 的核心思想是将模型组织成不同细节程度的层次，绘制时按视点需求进行适当选择。研究者们针对 LOD 技术进行了近 30 年的研究，目的是寻找更好的途径，在尽量少影响最终可视效果的前提下，使绘制量减到最少。但 LOD 方法有两个不可避免的问题，一个是对最终效果逼真度的影响，另一个是在 LOD 技术中不同层次的模型的过渡不平滑性。更为详细的 LOD 的介绍，将在本书的与绘制相关的章节中进行。

基于图像的渲染方法是减少绘制所需面片的另一种思路，它通过将场景做成图像，然后通过纹理映射的形式，应用于场景中。对于基于图像的渲染来说，绘制的面片数与场景的复杂程度无关，可以大幅度减少绘制的面片数。本书中所介绍的“动态纹理技术”是基于图像渲染的一种发展形式，在快速绘制的那一章里，将会有“基于图像的渲染方法”的详细介绍和描述。

**4. 图形硬件加速技术**

三维图形学研究的第四个方面是硬件加速技术。图形学的发展离不开图形硬件的发展，图形学硬件的发展反过来又推动图形学技术的发展。随着 GPU 编程

的普及，越来越多地开始运用 CPU/GPU 协作的并行计算模式。研究大规模应用的硬件加速技术成为一个新的热点。

图形硬件加速技术是本文所介绍和研究的一个重点内容，随着 nVidia 公司 GPGPU 架构以及近期的统一渲染架构的提出，目前越来越多的软件开始使用“图形硬件加速技术”，而且，图像硬件加速技术已不再是计算机图形学的专利，而是成为了各种应用的加速器，如高清电影编码解码、数据压缩与分析、图像处理、计算机视觉等方面都开始应用图形硬件加速技术。最近的 Adobe Photoshop 软件的 CS5 版本，已经开始应用图形硬件加速，硬件加速应用的火热程度由此可见一斑，其应用普及时代或许即将来临。本书将详细介绍图形硬件加速的原理以及实现方法，并在各章节的应用实例中穿插其具体应用方法。

## 1.2　计算机动画技术

随着计算机图形学和硬件技术的高速发展，越来越多的研究机构和商业机构加入到计算机动画领域，用计算机来生成各种以假乱真的虚拟场景画面和特技效果。在过去几十年里，计算机动画一直是人们研究的热点，在全球计算机图形学盛会 Siggraph 上，几乎每年都有计算机动画的论文和专题。计算机动画每年一度的学术会议“Computer Animation”和学术期刊《Journal of Visualization and Computer Animation》为专业人士进一步交流研究成果提供了平台。目前，计算机动画已经形成一个巨大的产业。

计算机动画是计算机图形学和艺术相结合的产物，是用于创作这些电脑特技的一项专门的技术，它伴随着计算机硬件和图形学算法的发展而发展，综合利用计算机科学、艺术、数学、物理学和其他相关学科知识，在计算机上生成绚丽多彩的虚拟真实画面。在 21 世纪的今天，计算机动画已经渗透进人们生活的每个方面。例如美国迪斯尼公司的众多动画片、日本的卡通片以及数字电影中的精彩场面，都给人们留下了深刻的印象。

1993 年电影《侏罗纪公园》利用计算机特效和动画技术制作的恐龙获得了当年的奥斯卡最佳视觉效果奖。1996 年的第一部完全计算机动画制作的电影《玩具总动员》上映，给电影制作开辟了一条新路。1998 年的电影《泰坦尼克号》通过计算机生成技术表现甲板上的人物，还有电影《终结者》中恐怖而又神奇的液态机器人等。这些电影都给我们带来了深刻的印象，计算机动画已经在许多应用领域证明了其非凡的潜力，也同样使我们领略到了计算机动画的高超魅力。

计算机动画主要研究物体的运动控制技术以及与动画有关的造型、绘制、合成等技术。目前计算机动画技术大致可以分为如下 8 类：

（1）参数关键帧技术；
（2）轨迹驱动技术；
（3）Morphing 和变形动画技术；
（4）过程动画技术；
（5）关节动画技术；
（6）基于物理的动画技术；
（7）剧本动画技术；
（8）行为动画技术。

这些技术保证了动画系统可以生成各种各样的复杂运动。

计算机动画根据其应用领域不同，可以分为二维动画和三维动画。二维动画研究对象为平面对象和二维形体，三维动画的研究对象为三维空间中的形体。按其实现技术来分，大致可分为关键帧动画、FFD 变形体动画、过程动画、关节动画和骨骼动画、基于物理的动画几方面。

本书的动作驱动技术部分，涉及计算机动画技术中的驱动原理，在相关章节中会给出更为详细的介绍。

## 1.3 虚拟现实技术

虚拟现实是做什么的？这是一个如此简单的问题，因为那么多的图形学元老以及专家们都曾给出过很明确的定义，如“利用计算机发展中的高科技手段构造的，使参与者获得与现实一样的感觉的一个虚拟的境界”；在地理信息学中，“虚拟现实是存在于计算机系统中的逻辑环境，通过输出设备模拟显示现实世界中的三维物体和它们的运动规律和方式”；从目前的应用来说，“虚拟现实是一种模拟三维环境的技术，用户可以如在现实世界一样地体验和操纵这个环境”这个定义更为确切。

虽然有如此繁复多样的定义，但到底哪一个描述更为确切？作者在这个领域的学习过程中，做过很多种类的工作，写过很多种类的程序，从图像处理中的直方图求解，到图像的卷积滤波边缘检测；从单个模型的建模和绘制，到虚拟场景的建立和漫游；从 OpenGL 中最基本的大茶壶的渲染，到骨骼动物或人物的运动控制；从三维物体的变形，到电磁场的可视化；从基本地形的 LOD 算法，到大规模地形的加速；从 CPU 编程到 GPU 编程；从风、烟、水等流体仿真，到衣服、软体的变形渲染，可能一页纸也未必说得完。

如果说，虚拟现实就是如其定义所述，使用计算机生成虚拟的场景，给人以类似于真实世界的体验，给人以沉浸感的体验，那么目前还未达到这样的目标。近些年的研究，在业界所能熟知的有很多，但为公众所有目共睹的却很少，公众

对于虚拟现实这个词仍然很陌生。但虚拟现实的研究仍不断发展，国内外有很多的研究机构都在从事于虚拟现实领域的研究，并同时激励作者也一如既往地进行其中的研究。

虚拟现实在图形处理方面所追求的两大目标是更快的处理速度和更高质量的图形结果，实时性和逼真性也是实现虚拟现实沉浸感特性的两大保证。随着计算机图形学所处理的规模越来越大，效果越来越好，虚拟现实是计算机图形学的一个代表领域，虚拟现实技术将人的视觉、听觉参与到信息处理的环境中去，将信息处理的过程建立在一个多维化的信息空间中，把便于计算机处理的单维信息改变为方便于人接受的各种表现形式。

虚拟现实技术中一个最近期的应用是电影《阿凡达》。《阿凡达》在 2010 年初算是出尽了风头，首部全 3D 效果影视，给大家带来了全新的视觉盛宴。其实，《阿凡达》的制作过程使用了大量的虚拟现实技术，从动作捕捉和面部表情捕捉到动作迁移，从计算机生成角色到角色动作驱动技术，从真实感场景绘制到角色群组仿真，《阿凡达》电影中唯美的场景和逼真的动作，几乎无处不体现着虚拟现实的迷人与神奇。图 1-1 所示为电影《阿凡达》的制作过程中精确的面部表情捕捉技术展示。

图 1-1　电影制作过程中面部表情捕捉技术⊖

## 1.4　群组仿真技术

回归本书的主题，群组仿真是做什么的？虚拟现实是一项综合集成技术，涉及计算机图形学、人机交互技术、传感技术、人工智能等领域，它用计算机生成逼真的三维视、听、嗅觉等感觉，使人作为参与者通过适当装置，自然地对虚拟世界进行体验和交互作用。在这个计算机生成的世界里，人物或者动物角色是这

⊖　图片来自于电影《阿凡达》宣传网站。

个世界中的重要组成部分，“有人或者有动物的地方，才是有生机的地方”，在角色仿真的应用中，甚至植物或者一切具备动态特征的物体，都可以看做是角色。

计算机角色仿真就是针对虚拟现实世界中的各类角色仿真方法进行研究的一个方向，它是虚拟现实与图形学领域的一个热点研究方向，与泛指意义上的虚拟现实有所区别的是，计算机生成角色仿真更重要的是研究场景中各类角色的运动仿真、行为仿真以及角色与静态场景的交互作用。

人、物以及其他各种运动角色是虚拟世界中的重要表现内容，而如何客观真实地表现这些角色是虚拟现实研究的一个核心问题。角色群组仿真是指在计算机生成空间（虚拟环境）中高效、逼真地展示大规模的角色——尤其是人物角色的运动行为。由于人们在物理、感知和交互等方面的基础性和主导性，使得对这些角色的运动仿真成为这个领域中的重要研究课题之一。

随着计算机特效技术的广泛应用，典型的例子就是数字电影特效在大众电影中的普及性使用以及游戏特效在网络游戏中的应用等，近几年人们的欣赏水平被极大提高，比如人们在看电影时追求大场面、高逼真的效果；比如在玩游戏时追求震撼、刺激的感受。这些都给角色仿真技术提出了新的要求，而这些电影特效的发展以及电子游戏的需求也在强力地推动着这一方向的研究。

群组仿真技术，也称为角色群组仿真技术或角色群组仿真技术，是指在虚拟场景中具备数量巨大的计算机生成角色时，针对角色仿真过程中所研究的一些特定的技术，如角色运动技术、大规模角色快速变形技术、大规模角色动作驱动技术、大规模角色快速绘制技术。

近年来，群组运动角色的仿真技术已逐渐扩展到模拟训练、教育教学等领域，利用角色仿真技术实现的社会活动模拟、紧急事件演习、虚拟文艺表演等应用开始崭露头角。如在 2008 年北京奥运会的开幕式上，每个运动角色有预设的路线和动作，几千个角色共同达到一个整体效果。这样的应用每一次排演的准备时间和排练时间都很长，对于导演的想象发挥形成很大的限制。借助于数字表演和仿真技术开发的虚拟编排系统，极大地缩短了演员之间的协调训练时间，可以提前发现问题，并修改其中的不足，达到最佳效果。

这些应用演绎出了人类无法通过直观思考获得的虚拟世界，为虚拟现实技术打开了一个新的应用之门，但这些应用都不可避免地面临一个问题——场景中的运动角色数量巨大，使得整个仿真过程变得极其艰难。与其他虚拟现实应用相比，角色群组仿真所需的场景规模大、涉及的角色数量多，从而造成计算量大、仿真困难等问题。角色群组仿真过程中所涉及的单项技术，在虚拟现实领域已经有研究者们提供了相应的解决方案，但随着计算机生成角色数量的增多，这些方案又面临着各种各样的局限和不足，这就使得研究者们又继续推动相关的研究。

## 1.5　计算机图形、图像、虚拟现实、群组仿真之间的关系

**1. 图形和图像的关系**

在很多与计算机相关的分类中，常常将图形和图像分为同一个类属。尽管这两类技术有着千丝万缕的联系，但图形和图像技术有着很大的区别。

图像和图形是在计算机中的两种不同表达形式，图形使用点线面体来表达整个世界，对于计算机屏幕上所展现的一切，均用顶点及顶点间的关系进行表达，图形表示与具体的计算机屏幕和计算机硬件无关。图像使用像素以及像素的颜色进行表达，图像中包含若干像素，每个像素具有其颜色，通过这些不同位置上的颜色的组合，从而形成一个场景。在图像中，每个位置上的像素之间没有必然的联系，而在图形中，点与点之间是有关系的。

对于图像来说，图像的复杂度与图像的内容无关，只与图像的像素数有关。而图形的复杂度，与图形所表示的物体大小无关，而与物体的细节程度有关，两者在不同情况下各有优缺点。图像学的处理主要包含图像平滑、锐化、图像分割、图像内容识别、图像编码等内容；图形学的处理主要包含图形建模、图形绘制、图形加速、图形硬件开发等内容。图形中常用到图像的技术，如纹理贴图、基于图像的渲染等，而图像中也有涉及图形的技术，如图像内容识别等。

**2. 虚拟现实与图形学的关系**

虚拟现实是使用图形图像、声学、视觉、触觉等手段，使人产生沉浸感的一种手段，虚拟现实技术汇集了计算机图形学技术、多媒体技术、人工智能技术、人机接口技术、传感器技术、实时计算技术等，它给用户更逼真的体验，人们可通过它来探索宏观世界和微观世界中不便于直接观察的事物。

在虚拟现实的各种手段中，逼真影像的产生是最难的部分，由于视觉在人的感知中所占的比重最大，而且人眼对于视觉图像的细微差异，特别是辨别性尤其敏感，而虚拟现实中的虚拟影像主要靠计算机图形学的方式来创建，因此在虚拟现实所涉及的技术中，计算机图形学所占的比重最大，图形处理的过程也是最为复杂的，因此计算机图形技术是最为研究者们所关注的方面，在很多研究机构中的虚拟现实实验室，其主要研究内容就是计算机图形学。

**3. 虚拟现实与群组仿真的关系**

群组仿真是虚拟现实的一个研究子领域。群组仿真是指在同一个场景中存在数量巨大的运动角色，并且每个群组中的角色运动存在一定的相似和重复的仿真形式，则这些角色的运动可被分为群组运动。计算机群组仿真以计算机生成角色虚拟场景中的人物、动物和植物等仿真实体，在仿真过程中，需要使用模型变形、动作驱动和快速绘制几方面技术。计算机群组仿真技术可以支持虚拟演习、虚拟训

练等应用，在大型文艺编排、团体操、大型操等大型演练项目中也经常使用。

## 1.6 群组仿真技术的应用实例

虚拟现实技术的发展和进步为社会发展提供了巨大的推动作用，目前许多实际项目在正式应用之前，都进行计算机仿真，以验证项目的可行性和实用性。得益于同行人士的努力，虚拟现实技术的现实应用已经进入万千大众的视野，而且也已引起大众对其的极大兴趣。

下面介绍几种公众所能够接触到的虚拟现实领域的技术应用。

**1. 军事推演中的应用**

虚拟现实与角色仿真技术最早的推动需求应该是来自于军事领域。以往的军事演习，都会耗费巨大的人力物力，亟需寻找一种便捷的、低成本的推演方式，而传统的推演方式，是使用军事地图进行的兵力布置推演，直观性和交互性差，难以还原真实的战场环境。

虚拟现实技术可以使诸军种联合虚拟演习建立一个“虚拟战场”，使参战双方同处其中，根据虚拟环境中的各种情况及其变化，实施“真实的”对抗演习。在这样的虚拟作战环境中，可以使众多军事单位参与到作战模拟来中，而不受地域的限制，从而大大提高了战役训练的效益，还可以评估武器系统的总体性能，启发新的作战思想。

在这种军事推演应用中，“计算机生成兵力”是其中一个重要的组成成员，每个作战单位需要大量的虚拟角色，以实施“真实的”作战规划，因此军事推演中的角色群组仿真技术是一个先驱型的应用实例。

**2. 应急推演中的应用**

防患于未然是具有一定危险性行业（消防、电力、石油、矿产等）的关注重点，如何确保在事故来临之时做到损失最小，定期地执行应急推演是一种传统并有效的防患方式，但其弊端也相当明显，投入成本高，每一次推演都要投入大量的人力、物力，大量的投入使得其不可能进行频繁性的执行。虚拟现实的产生为应急演练提供了一种全新的开展模式，将事故现场模拟到虚拟场景中去，人为地制造各种事故情况，组织参演人员做出正确响应。这样的推演大大降低了投入成本，提高了推演实训时间，从而保证了人们面对事故灾难时的应对技能，这样的案例已有应用。

在应急推演的应用中，会使用到大量的人物或者其他运动角色，而其中每一个角色又都具备各自的行为，因此给应急推演仿真带来巨大的困难。图 1-2 所示为利用图像渲染技术渲染的应急推演场景[4]。

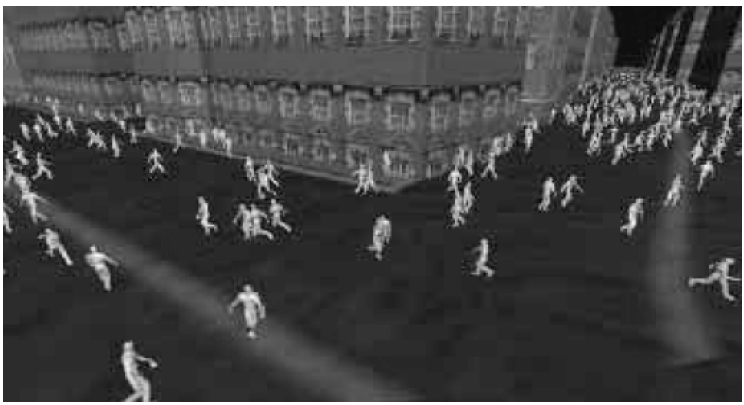


图 1-2　利用图像渲染技术渲染的应急推演仿真

**3. 虚拟社区中的应用**

数字城市曾经是一个很热门的研究方向，城市的数字化也是一个现代化发展的必然结果，在城市规划和城市建设中，利用虚拟现实技术，基于真实数据建立的数字模型组合成虚拟社区，严格遵循工程项目设计的标准和要求建立逼真的三维场景，对规划项目进行真实的“再现”，大大提高了项目的评估质量。

在旅游景区，也可以利用角色群组仿真技术生成大量的计算机角色，仿真旅游人群活动，从而辅助制订应急预案。体育运动场中的大量观众也是一个角色群组仿真技术的常用应用场景，图 1-3 所示为一个体育场馆的观众仿真实例。

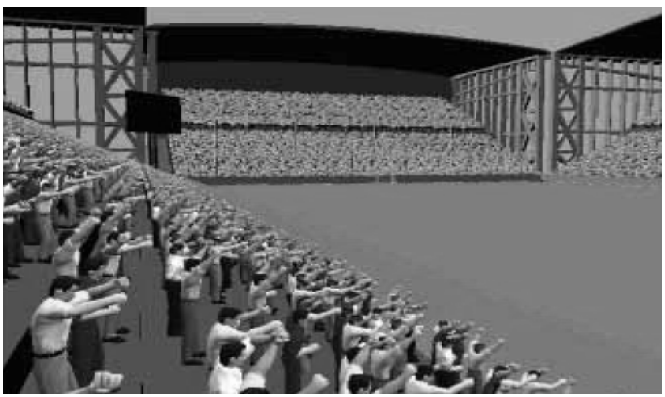


图 1-3　利用点渲染技术渲染的体育场馆

**4. 电影制作中的应用**

电影制作从来就少不了特效的参与，数字化特效不仅仅能实现眼花缭乱的武打动作、不可思议的艺术效果，还可以实现千军万马的战争场面。在很多电影中都会需要角色群组仿真的技术，电影《英雄》中一望无际的秦军，就使用了数字

特效，而在电影《指环王 3：王者归来》中，Weta 工作室制造了很多战争人物的实体模型，然后让动画师把它们扫描进计算机，做出计算机动画人物，每个虚拟的动画人物都有自己的打斗动作、招式，让观众分不清哪个是真人哪个是动画人物。最终在电影场景中有 20 万个计算机制作出来的虚拟人同时进行战斗，其实扮演兽人的演员只有 100 个，马也没有超过 500 匹。图 1-4 所示为《指环王 3：王者归来》电影中宏大的战争场面。

图 1-4　电影《指环王 3：王者归来》中宏大的战争场面㊀

**5. 网络游戏中的应用**

除电影外，许多三维网络游戏的开发过程也大量应用了群组仿真的技术。自从 2001 年第一款网络游戏开始在国内出现，游戏产业的发展得到了迅猛的发展，至 2008 年为止已达到了 400 亿元的产业规模，游戏也从传统的二维游戏发展到了三维游戏。三维游戏的华丽界面，以及便捷的操作方式，相信给各位玩家都留下了深刻的印象。游戏中大量的 NPC（Non-Player Character，非玩家角色），或多或少地使用角色仿真的技术实现，而在一些典型的人物类游戏（如著名足球游戏 FIFA）中，则是角色群组仿真应用的典型代表。图 1-5 所示为著名足球游戏 FIFA 中的逼真场景。

**6. 数字表演中的应用**

随着大型晚会和大型运动场面的编排需求，数字表演慢慢走向大众视野，2008 年北京奥运会开幕式同样也是群组仿真技术的一次应用，北京奥运会已过去两年多的时间，但相信开幕式的精彩大家印象仍然深刻，奥委会主席罗格称之为“无与伦比”。虚拟编排技术为导演发挥更大胆的想法和灵感提供了支持；同时也为实际排练提供了可操作的数据，大大提高了排练效率，使得如此大规模的排练成为可能。图 1-6 和图 1-7 所示分别为 2008 年北京奥运会开幕式中的节目“太极”的数字表演仿真效果和实际自然表演效果图。

㊀ 图片来自于网络《指环王 3：王者归来》宣传海报。

图 1-5　著名足球游戏 FIFA 中的逼真场景[7]

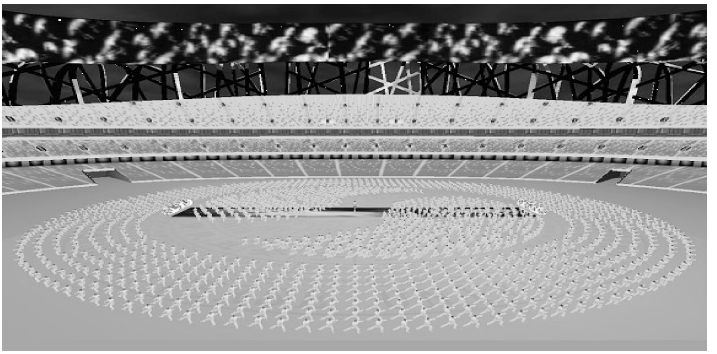


图 1-6　2008 年北京奥运会开幕式中的节目“太极”的仿真效果㊀

图 1-7　2008 年北京奥运会开幕式中的节目“太极”的自然表演效果㊁

㊀ 图片来源：http：//www. bit. edu. cn/col183/article. html? id = 26248。

㊁ 图片来源：新华网 http：//www. xinhuanet. com。

上述列举了几种虚拟现实技术和群组仿真技术的典型应用，但此类技术所应用的范围绝不仅仅限于这些，在诸如交通指挥、医疗服务、教育教学、培训演习等领域，无不活跃着虚拟现实技术和群组仿真技术的影子。在后续的章节中我们就来探究群组仿真相关的技术原理，以及讲述作者本人在这方面的研究成果。

## 1.7 本章小结

本章针对计算机群组仿真技术相关的各项技术进行了简单介绍，引出计算机群组仿真技术领域，总结前人在相关领域的知识和现状。其中包括了计算机图形学技术、计算机动画技术、虚拟现实技术以及群组仿真技术，针对这些技术介绍了它们的概念、主要技术点等知识，并分析了这些技术之间的区别和联系。

对于计算机图形学，讲述了从二维图形学到三维图形学的发展，以及它们之间的区别和联系。对于虚拟现实技术，着重分析了它的概念和定义。对于群组仿真技术，本章介绍了一些近期的应用以及其中的技术点。

# 第 2 章　群组仿真的关键技术

虚拟现实（Virtual Reality，VR）技术[1,2]能够模仿人的视觉、听觉、触觉等感知功能，使人可以体验虚拟现实环境，并能与该环境相互作用。虚拟现实的最终目的是提高人的认知能力，促进人与环境的交流，更深入地开发人类的智慧。目前，虚拟现实技术是一个热门研究课题，在许多领域已经得到了成功的应用，如军事领域中的虚拟演练、虚拟装配、武器装备的先期验证、武器性能评估、武器系统仿真等。

计算机生成角色仿真是虚拟现实与图形学领域的一个热点研究方向，而电影特效的发展以及电子游戏的需求正强力推动着这一方向的研究。当人们沉醉于这些动人心弦的画面时，其实已经不知不觉地与计算机图形学有了亲密接触。如今特效的应用越来越普遍，已经成为一种必不可少的手段，而计算机图形仿真技术的价值也渐渐被人们所重视。

近年来，虚拟现实技术逐渐向大众化应用发展，尤其包含运动角色的仿真技术已逐渐扩展到模拟训练、教育教学等领域，虚拟现实技术与社会应用的结合也越来越多，利用虚拟现实技术实现的社会活动模拟、紧急事件演习、虚拟文艺表演等应用开始崭露头角。在 2008 年北京奥运会开幕式上，采用数字表演和仿真技术开发的虚拟编排系统，为节目的创意呈现和组织排练做出了开创性的贡献，极大地缩短了演员之间的协调训练时间，这些应用演绎出了人类无法通过直观思考获得的虚拟世界，为虚拟现实技术打开了一个新的应用之门。

人、物以及其他各种运动角色是虚拟世界中的重要表现内容，而如何客观真实地表现这些角色是虚拟现实研究的一个核心问题。角色群组仿真，也称为大规模运动角色仿真，是指在计算机生成空间（虚拟环境）中高效、逼真地展示大规模的角色——尤其是人物角色的运动行为。由于人们在物理、感知和交互等方面的基础性和主导性，使得对这些角色的运动仿真成为这个领域中的重要研究课题之一。

与其他虚拟现实领域中的仿真应用相比，计算机角色群组的仿真所需的场景规模大、涉及的角色数量多，从而造成计算量大、仿真困难等问题，这些困难具体体现在如下几方面：

（1）目前的变形算法大多适用于单个模型的变形，随着变形角色数量的增多，大量角色的实时变形计算无法实现；

（2）真实感运动数据获取比较困难，人物以及动物等角色具有最复杂的动作和最不确定的行为模式，而目前的动作数据格式各不相同，不同途径获得的动作

数据共享性差；

（3）场景中角色绘制的实时性难以保障，随着角色数量的增多，角色的绘制总时间迅速增大，这对可实时仿真的角色规模造成很大的限制。

由于上述困难，如何快速、便捷地使大量角色较真实地动起来成为一个难题。针对这些困难，本书以二维和三维角色模型的快速变形技术为中心开展研究，探讨大规模运动角色的实时仿真方法，为各类大规模动态场景的绘制提供可用的技术支持。本书的主要研究内容包括快速的二维角色变形方法、快速的三维角色变形方法、真实感动作驱动方法和大规模角色的快速绘制技术。

## 2.1 运动角色仿真流程

真实感和实时性是所有虚拟现实仿真应用追求的两个目标，两者相互共存、相互制约。在群组运动角色仿真中，真实感体现为角色变形结果的真实感和运动动作的真实感，实时性体现为角色变形的实时性和绘制的实时性。随着计算机软、硬件的发展，人们对计算机生成场景的真实感需求不断提高，高度真实感的处理与绘制需要大量的计算；另一方面，满足交互需求的实时场景绘制是保证虚拟环境沉浸感的关键性环节。虚拟现实对真实感和实时性的要求，决定了处理便捷、渲染快速、效果逼真的仿真方法是其重要研究内容。

随着虚拟现实技术的发展，数字化运动仿真技术日趋成熟，其应用也被迅速推广，大规模群组运动角色仿真技术开始成为虚拟现实领域新兴的研究热点，同时，仿真领域和其他计算机应用领域相互交叉、相互融合成为了一种趋势。在大量非仿真专业领域的应用系统中，运动角色快速仿真技术得到了广泛应用，如团体操、大型操、文艺表演、阅兵演习、会场排练等大型队形编排项目。这些项目常使用虚拟现实技术进行创意呈现，以及时发现和修改其中的问题，从而达到最佳效果。这种现代化手段不仅帮助提高创意空间，而且节约人力物力、提高效率。“以人为本”，以人为主要的研究对象，是以后的发展趋势，随着这一趋势的发展，仿真系统中运动角色仿真的重要意义也必然越来越得到扩展和加深。

对于运动角色的群体行为仿真，法国的 Eric Bouvier[3] 使用粒子模拟仿真对象，利用粒子系统建立人群运动模型。Franco[4] 利用图像渲染技术实现虚拟社区仿真。瑞士 Soraia[5,6] 等人提出 Vi Crowd 模型，定义了虚拟角色群体仿真的三个不同层次集合：群体、团队和个体。在国内的相关研究领域中，中科院计算所的王兆其教授[7] 等人研究了关于大规模虚拟人的仿真技术。

实现大规模角色群组运动仿真，需要解决两类关键技术问题：

（1）研究并建立单个运动角色的动态仿真模型，即如何更真实地完成对运动过程的模拟。这需要有两个先决条件：接近于真实世界的运动数据和快速的变形

算法，人眼对运动过程差别具有高度敏感性，稍有差别的运动动作就将影响运动角色仿真的视觉效果。

（2）研究大规模群组运动角色的快速绘制，即如何将大规模的角色群组运动逼真地展现出来。角色群组运动仿真涉及角色数量多，需要实现大量角色的实时绘制，以呈现运动角色的各种姿态和整体运动效果。

大规模角色群组运动仿真的技术结构如图 2-1 所示。

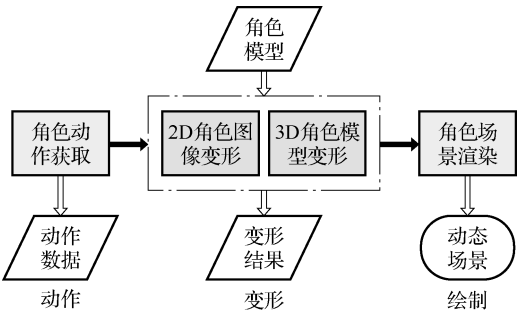



图 2-1　大规模角色群组运动仿真的技术结构

在整个仿真流程中，角色动作获取过程负责提供接近于现实世界中的动作数据；模型变形部分提供快速的适用于多角色同时变形的方法，其中根据所变形的对象分为二维角色变形和三维角色变形；角色场景渲染完成大规模群组运动角色的实时绘制。这几部分以模型变形技术为中心，分别解决了运动动作配置繁琐、变形方法不适应大数量级的角色变形和大规模角色群组的绘制实时性难以保障等问题。

大规模角色群组运动仿真涉及的相关技术如图 2-2 所示。

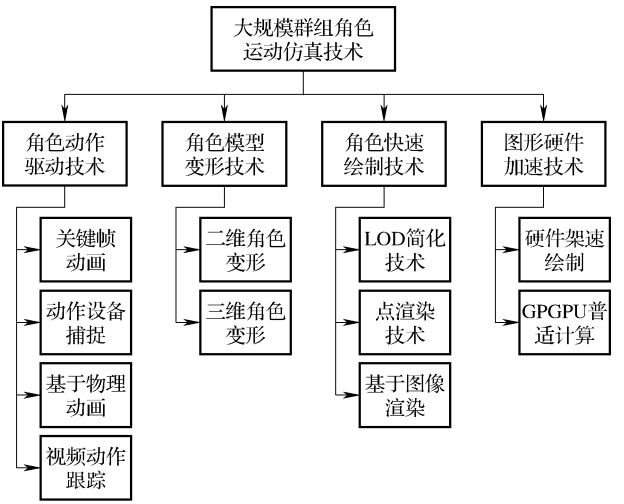



图 2-2　大规模角色群组运动仿真涉及的相关技术

（1）角色动作驱动技术。传统的运动数据建立方法包括关键帧动画[8]、专业设备运动捕捉[9]和基于物理的动画[10]三类，近年来基于视频的动作获取[11]也已成为计算机视觉研究中的热点问题。

（2）角色模型变形技术。传统的模型变形包括 Morphing 变形[12]、FFD（Free-Form Deformation）自由体变形[13]、基于骨骼的变形[14]和基于物理的变形等几种方法，最近基于覆盖网格的图像变形[15]方法和细节保持（Detail-Preservation）[16]的模型变形方法已成为主流。

（3）大量角色快速绘制技术。虽然目前计算机性能得以很快的提高，但随着场景规模的增大，大量运动角色的实时绘制仍然存在困难。LOD 细节层次技术[17]、点渲染技术[18]和基于图像的渲染技术[19]等都被用于解决该问题。

（4）图形硬件加速技术。图形硬件由于可编程性和并行性成为一种必要的加速手段，图形硬件加速途径主要包括两类：一类是图形加速绘制[20]，主要通过 GPU 改变几何模型的顶点或像素属性；另一类是 GPGPU（General Purpose Computaion on GPU，基于 GPU 的通用计算）普适计算[21]，用于通过 GPU 执行通用计算功能。

## 2.2 角色动作驱动技术

为了更好地模拟人物或动物运动，获取更接近于真实的动作数据尤为重要。在计算机运动角色仿真应用中，最主要的动作获取对象是人，人物作为虚拟场景中的角色，一直是研究者感兴趣的目标，其应用也与日俱增，特别在游戏、电影、军事、体育等领域得到了越来越广泛的应用。对人体运动仿真技术的研究，可以大大提高虚拟世界的表现能力，有效检验各种活动的设计效果，在电影《终结者Ⅱ》和电影《侏罗纪公园》中就大量地应用了设备动作获取的技术。人体具有 200 个以上的自由度和非常复杂的运动，人体的肌肉随着运动而变形，人的个性、表情等千变万化，可以说，人体运动仿真是计算机仿真中最富挑战性的课题之一[22]。

在虚拟人体的工作研究中，比较著名的是宾夕法尼亚大学 Badler 领导的人体建模与仿真中心[23]和蒙特利尔大学 Thalmann 带领的在洛桑的 LIG 实验室[5,24,25]。传统的运动数据建立方法主要包括关键帧动画[8]、专业设备运动捕捉[9]、基于物理的动画[10]和基于视频的动作跟踪[11]四类方法。

关键帧技术指定关键帧动作状态和该关键帧发生的动作时间，使用各种插值算法计算关键帧之间的中间动作状态，实现角色的连续运动。这种方法的角色动作依赖于用户的设置或输入，良好的动作序列需要用户仔细地配置和调整参数，以尽量避免出现活动动作生硬板滞、明显卡通化的问题。

使用专业运动捕捉设备获取动作数据[9]是目前唯一能够直接建立真实的人体运动的技术，具有真实感强的显著特点，在娱乐、电影和游戏应用中广泛应用，目前的大多数运动动作数据也是使用专业设备捕捉获得的[11]。2005 年 Naksuk[26]使用运动捕捉设备获取人体运动数据，并通过关节点间的位置关系将运动迁移到机器人上。这类方法可以获得真实准确的运动数据，但它需要专用的捕捉设备，且捕捉过程依赖于被捕捉对象的运动能力。

基于物理的仿真[10]是通过正运动学或逆运动学，计算角色运动过程中的动作姿态，该方法不受设备限制也不受目标对象的运动能力限制的一种仿真方法，但它需要针对每一套动作设置计算公式，而且该方法对参数的变化很敏感，需要仔细地设置各个参数，才能获得良好的动作效果。

基于视频的动作捕捉方法[11]是通过录制现实世界中的人物或动物运动，结合所跟踪目标的形状特征和纹理信息，跟踪角色各标定点的位置，从而得到运动数据的方法。这种方法有一定的限制，如数据精度不如设备捕捉好，但拥有获取容易、成本低、动作连贯性好、视觉效果真实的特点。2003 年 Yoshimoto[27]使用了视频图像提取方法估算特征点的位置来获取人体姿态。Wu[28]通过基于视频的学习方法实现效果十分逼真的鸟类飞行模拟。2005 年 Kehl[29]使用多台摄像机捕获人体标定关节点的运动轨迹，建立了人体骨骼模型的运动序列。2006 年 Li[30]系统地总结和介绍了基于视频的人体运动捕捉研究的技术和方法，2006 年 Wan[31]使用 3D Graph-cuts 和人体形状模板匹配估算视频中人体姿态。2007 年 Pei[32]基于 ISO-map 的非线性降维和 K-means 聚簇方法，从演讲视频中提取嘴部动作，并通过控制形状向量实现面部变形。

## 2.3　角色模型变形技术

长期以来，模型变形一直是计算机图形学领域的热点内容，与工程仿真中严格的力学变形不同，计算机图形学中模型变形普遍要求响应速度和逼真度的均衡，如何在满足实时要求的前提下实现尽可能逼真的变形效果是研究的最终目的。

模型表示的选择对变形方法至关重要，模型表示的不同对变形计算速度、变形结果真实性都有重要的影响。静态模型表示方法的分类如图 2-3 所示。

根据所描述的目标物体的类别可分为二维图像、三维表面模型、三维实体模型和过程模型，在计算机图形学领域中，二维图像和三维表面模型是最常使用的模型，三维表面模型有多边形、参数化曲线曲面和内隐方程表面等几种表示方法，而多边形表示法又是最常用的。Cretu[33]提供了一个 3D 物体建模方法的总结，讨论了各种建模方法的优劣及其建模结果的质量。Collins[34]总结了角色动画

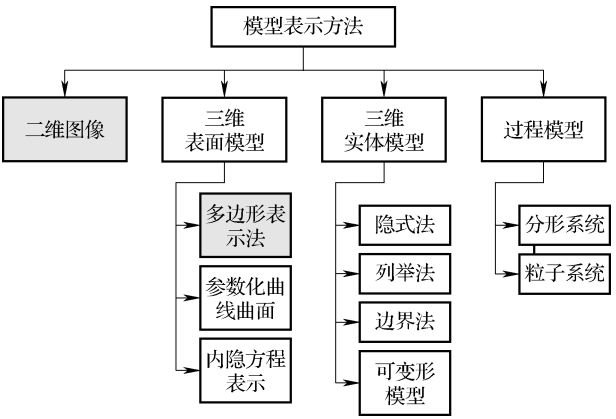



图 2-3 静态模型表示方法的分类图

中所使用的模型及变形方法，对多边形表示、参数表面、内隐表面及其各种变形方法均做了较详细的讨论。Casale[35]和 Gibson[36]分别对实体模型和可变形物体的建模方法进行了总结。

合理逼真的变形结果和快速高效的变形计算是变形系统追求的目标，一个优秀的变形系统允许用户只用简单的操作即可得到逼真的变形结果，为此国内外学者进行了大量研究，提出了大量变形方法。变形方法按目标对象可分为二维图像变形和三维模型变形；按变形技术可以分为 Morphing 变形[12]、FFD 自由体变形[13]、基于骨骼的变形[14]和基于物理的变形。其中二维图像变形一般使用 Morphing 或 FFD 变形，2005 年 Igarashi[15]使用几何网格覆盖图像，通过网格变形实现图像变形，使用图形的方法解决图像的问题；三维模型变形一般使用 Morphing、FFD 和基于骨骼的变形方法，近年来细节保持的变形方法已成为主流，它将模型分离为基础模型和细节模型，然后分别进行处理，可以保持模型表面的高频细节。

### 2.3.1 传统变形方法

在传统的变形方法中，Barr[37]的整体和局部非线性变形是这一方面最早的工作，他推广了传统的造型操作，将变换表示为位置的函数，把变形操作分为 Tapering、Twisting 和 Bending 几类。Lewis[38]将各类变形方法统一用 Pose 空间变形解释，并使用形状插值和径向基函数进行变形计算。

Morphing 是最早的变形技术，对于两个不同的模型，通过定义特征点和特征线建立两个模型之间的对应关系，然后插值对应属性和位置，可以获得从一个模型到另一个模型的中间过渡模型。由于该技术能够产生奇特的视觉效果，曾被广泛应用于三维造型及计算机动画系统之中。比较成功的算法有基于网格的变形算

法[39]和基于域的变形算法[12]。2000 年 Alexa[40]使用插值方法实现了刚体的变形。2001 年 Sloan[41]使用线性径向基函数插值关节体和人脸样本产生连续的形态。2003 年偶春生[42]通过对应角色有效区域边界和边界点位置控制，实现二维角色图像的 Morphing 过渡。2007 年 Martin[43]引入黎曼几何距离概念，计算给定模型的等距变形结果，最后使用形状插值获得任意状态的变形结果。2008 年，Guo[44]通过自动计算形状拓扑并匹配轮廓顶点，在 Morphing 过程中保持形状边长及其局部、全局和边界属性。

FFD 是一种经典的自由变形方法，最早由 Sederberg 和 Parry[13]在 1986 年提出，通过对物体所嵌的空间变形实现任意拓扑结构的内嵌物体的变形，是一种统一有效的与物体表示无关的整体变形方法。Coquillart[45]提出以非平行六面体作为物体所嵌的空间的 EFFD（Extended FFD）方法。Coquillart 和 Jancene[46]提出可实现交互式 FFD 动画变形的 AFFD（Animated FFD）方法。Kalra[47]提出 RFFD（Rational FFD），并用其实现了面部表情的动作仿真。Lamousin[48]等人实现了基于 NURBS 的 NFFD（NURBS-based FFD）方法，提供了一种功能更强的变形控制方法。Celniker[49]使用 FFD 方法实现了曲线和表面的有限元变形仿真。MacCracken[50]使用点阵网格划分变形空间并对该空间进行 FFD 变形。Singh[51]以曲线作为变形控制点，通过变形物体所在的空间实现模型变形。2000 年 Chua[52]提出一种硬件加速的 FFD 变形方法。2002 年 Milliron[53]较详细地分析了各种几何 Warp 和变形方法，并将这些弯曲和变形操作统一到同一个框架中。2003 年 Draper[54]提出了一种使用手势控制 FFD 变形的方法。Faloutsos[55]实现了动态 FFD 变形。

基于骨骼的变形方法，通过控制骨骼关节点的位置，获得需要的角色姿态。Stalpers[14]使用多边形网格模拟骨骼，对几何模型进行变形。2000 年 Lewis[56]使用空间表示方式统一形状插值和骨骼驱动的变形，通过 pose 空间的映射进行物体变形。2002 年 Capell[57]根据角色骨骼结构把图像划分为若干子区域实现交互式角色变形，Paul[58]对每个骨骼关节点计算其变形的主成分影响因子，实现基于图形硬件的非线性皮肤变形。2003 年 Mohr[59]使用自学习方法由一系列姿态实例计算骨骼模型的关节点权值，通过调整变形模型参数实现快速变形。2005 年 Sumner[60]通过学习给定网格模型的变形模式，采用逆运动学计算实现变形网格的受力变形。2006 年 Yan[61]使用边缘控制与三角辅助网格覆盖，实现基于骨骼的图像角色变形。2007 年 Kevin[62]通过逆运动方程计算关节点位置，实现分辨率无关的网格变形。对于角色的骨骼建立方法，2006 年 J. M. Lien[63]详细描述了几种合理建立骨骼模型的方法。

由轴线控制的物体变形可以看作是一种基于骨骼变形的变体，该方法对于物体上的每一点，计算该点相对于轴线的局部坐标，根据变形后的轴曲线计算出顶点的新位置。Chang[64]提出了基于曲线迭代仿射变换的变形方法。Lazarus[65]提出

了一种直观的轴变形方法，彭群生[66][67]提出了基于弧长保持的轴线变形方法，完善了这一模型。在最近几年里，2006 年 Yang[68]使用三角形网格的中间曲线作为控制线，通过曲线控制人物或艺术角色变形。2006 年 Nicu[69]讨论了曲线骨骼的属性、应用和基于曲线骨骼的变形算法。2007 年 Shi[70]采用逆运动学计算网格变形。2002 年 Capell 等人[71]将模型划分为线骨骼和控制网格体两层结构，角色嵌入网格体中变形，但无法实现如扭转等复杂的动作。2006 年石教英等人[72]采用线骨骼、三维中间层和控制网格体三层模型结构，实现了骨架驱动的快速人体动画。

### 2.3.2 最新的模型变形方法

在最近几年里，研究者们针对角色图像的变形提出了使用几何网格覆盖变形角色图像，通过变形网格形状实现图像变形的方法，使用图形的方法解决图像的问题，如 2005 年 Igarashi[15]使用三角形辅助网格覆盖 2D 角色模型，使用最小二乘法求解网格自由顶点的位置获得变形结果。2006 年 Schaefer[73]间隔提取像素点作为辅助网格，使用最小二乘法求解网格自由顶点实现角色图像变形。2006 年 Weng[74]和 Yoshioka[75]都将影响因素表达为方程组，分别使用非线性最小二乘法和约束最小二乘法求解该方程组，获得自由顶点坐标和图像变形结果。如图 2-4 所示为 Igarashi[15]、Schaefer[73]和 Weng[74]三种方法分别使用图像变形的覆盖网格示例。

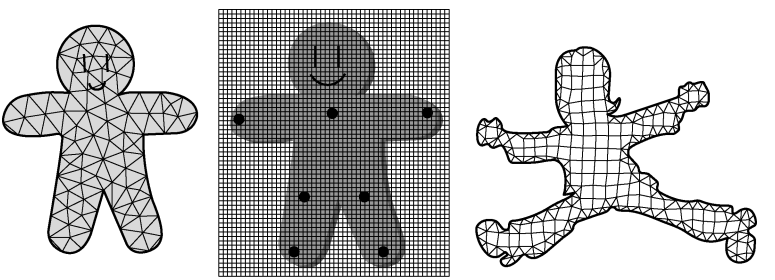



图 2-4 图像变形的覆盖网格示例

a）Igarashi[15]的网格 b）Schaefer[73]的网格 c）Weng[74]的网格

细节保持的三维模型变形技术越来越受到重视。细节保持可以减少高频细节信息丢失，从而解决变形处理中的尖角凸起细节丢失和模型表面平缓化的问题，提高变形逼真度。在细节保持的变形方法中，原始模型被分离为一个变化率较低的基础模型和若干个高变化率的细节模型，在变形中只对基础模型进行处理，变形后再将细节模型合成到基础模型上，由于细节模型不参与计算而减少了细节

丢失。

细节保持的变形中常使用 Laplacian 坐标[76]和 Poisson 坐标[77]来实现。2003 年 Alexa[78]和 Lipman[79]基于差分坐标实现了模型过渡和变形，在 2006 年 Alexa[80]又探讨了离散 Laplace 和 Poisson 坐标的网格变形方法。2004 年，Sorkine[76]和 Yu[77]分别将 Laplacian 坐标和 Poisson 坐标用于表面网格模型编辑，为细节保持的模型变形提出了一种新的思路。2007 年 Nealen[16]和 Wu[81]分别给出了基于素描的和基于骨骼的细节保持变形方法。

除细节保持外，体积保持也是一种重要的模型属性保持的方法，对提高变形结果真实感具有重要作用。Kobbelt[82]采用法向位移，对原始模型顶点在基础网格上的对应点进行位置编码，避免了对原始模型重采样造成的误差过大问题，但该变形方法会导致物体的体积变化。2003 年 Botsch[83]引入位移容积（Displacement Volume）概念，对局部棱柱体积进行位置编码，实现物体变形过程中的体积保持，但该方法计算量大、无法用于实时交互。2005 年 Choi[84]使用旋转角分解实现细节保持的大角度模型变形。2006 年 Huang[85]将模型映射到基础网格上进行变形，采用基于表面代替以往的基于空间的变形，实现简化模型的插值。2007 年 Oscar[86]使用控制变量集合表达变形，将变形模型简化为 isoline，由 isoline 组成类似骨骼的虚拟控制结构，通过插值 isoline 和原模型实现低分辨率的模型变形。

## 2.4　角色快速绘制技术

角色绘制根据对实时性的要求可分为非实时绘制和实时绘制[87]。非实时绘制侧重于角色模型的真实感，为了追求真实感不惜增加时间开销，主要应用于电影、电视、动画等方面。实时绘制侧重于交互性和绘制速度，真实感被放在次要位置，主要应用于虚拟战场、实时仿真、紧急疏散演习、城市管理和群体性行为仿真等领域。

计算机能够达到实时绘制取决于两点：

（1）所采用的计算机平台具有很高的运算速度；

（2）所绘制的目标模型拥有尽量少的面片数。

在固定帧速率条件下，一台计算机所能绘制的原始几何面片数是有限的，因此简化绘制目标的复杂度是实现复杂模型实时绘制的主要方法，即通过减小计算量来实现实时性能。常用的模型简化方法有 LOD 简化技术和基于图像的渲染技术。

LOD 技术将一个模型简化为多个不同的三角形分辨率层次，在实际绘制过程中，根据对视觉的贡献率选择不同的分辨率层次。目前 LOD 技术有离散 LOD、

连续 LOD、视点相关的 LOD 和层次 LOD 几种[17]。离散 LOD 将模型预处理为互相独立的细节层次模型，连续 LOD 在运行时计算 LOD 细节模型，视点相关的 LOD 可能会对一个单独的物体使用几个不同的 LOD 层次，层次 LOD 将若干个小物体组合为一个整体进行简化而不是对每个物体进行简化，解决了小模型的显示问题。Heok[17]和何晖光[88]分别给出了一个 LOD 方法的综述和网格模型简化的综述，潘志庚[89]给出了多细节层次模型自动生成技术的综述，Chadwick[90]给出了一种动画角色的层次化创建方法，Hoppe[91,92]给出了一种不规则网格的离散简化与渐进式简化方法，Cignoni[93]比较了各种网格简化算法的优劣，Ogren[94]给出了一种基于连续 LOD 的实时地形渲染算法，2005 年张昌明等[95]给出了一种基于边折叠的多边形网格模型简化算法，在减少屏幕误差的前提下提高图形绘制速度。

基于图像的绘制（Image-Based Rendering，IBR）技术[19,96,97]又分为无几何、基于隐式几何和基于准确几何三类。无几何图像渲染使用全景图来构建虚拟环境，在虚拟环境中漫游相当于选择不同的全景图。基于 Billboard 的渲染方法则是一种基于隐式几何的图像渲染方法。而传统的纹理映射方法是一种依赖于准确的几何模型但只需要少量图像的方法。图像渲染方法的优越性是生成图像的质量与场景的复杂性无关，不必通过使用专门的硬件加速就能获得很强的真实感和实时的交互速度。

Billboard 是一种常用的复杂模型简化和渲染技术，它将模型投影成一幅图像，使用以该图像作为纹理的平面板代替模型进行快速渲染。BillBoard 技术与 Alpha-Texture 结合，可以显示如烟、雾、火、爆炸、云[98,99,100]、树木[101]等多种难以描述的物体。这类物体采用多边形建模需要大量的多边形，而采用 BillBoard 技术只需少量的多边形，但 Billboard 只能渲染静态的图像，不能在三维场景中展现模型的动态效果，这样就很大地限制了它的应用范围。Schaufler[102,103]提供一种动态替代板技术（Dynamic Impostors）实现快速绘制，通过重新渲染原多边形模型更新 Billboard 的纹理，但重新渲染多边形模型需要消耗很多时间。Max[104]通过预计算获得离散视点位置的纹理图像，然后通过插值使这些纹理图像生成连续的纹理。2004 年 Anton[105]通过图像简化，使用 15 ~ 50 个纹理三角形表示树木模型，采用离散 LOD 方法渲染森林场景。2007 年 Deng[106]根据树木模型建立多分辨率 Billboard 云层次，并通过视点距离与重要性选择层次实现交互式植物渲染。

## 2.5 图形硬件加速技术

GPU（Graphics Processing Unit）是由 NVIDIA 公司首先提出来的，1999 年 8 月 NVIDIA 公司发布了全球第一颗 GPU 显卡——GPU-GeForce-256，它由两个模块

组成：一个是顶点转换和顶点光照处理器，可以进行 32 位浮点数运算；另一个是像素着色流水线（或称为片元着色流水线），可以进行定点数运算。2001 年 NVIDIA 推出了第一款拥有可编程顶点着色器的 GPU-GeForce 3，它使用了可编程的顶点处理器和不可编程的像素处理器（或称为片元处理器），两者都能完成 32 位单精度浮点运算。2006 年的 GeForce 8 系列 GPU 率先采用了统一渲染架构，以通用渲染单元替代了原来分离的顶点着色单元和像素着色单元。2007 年 CUDA 正式发布，引发了 GPU 通用计算的革命。

近年来，GPU 正在以大大超过摩尔定律的速度高速发展，极大地提高了计算机图形处理的速度和质量，它不但促进了图像处理、虚拟现实、计算机仿真等相关应用领域水平的快速提高，同时也为人们利用 GPU 进行图形处理以外的通用计算提供了良好的运行平台。

GPU 应用领域的拓宽与其硬件发展有着极大关系。GPU 自 1999 年首先由 NVIDIA 公司提出后，就其发展的速度而言，是 CPU 更新速度的 3 倍。2004 年，NVIDIA GeForce 6800 Ultra 处理器峰值速度可达 40GFLOPS，对比同时代的 Intel Pentium4 3.0GHz，采用 SSE2 指令集也只能达到 6GFLOPS。NVIDIA 发布的 GeForce 8800 图形处理器集成了 6.8 亿个晶体管，拥有 128 个流处理单元，其峰值运算能力超过 340GFLOPS，而与此同时 Intel 的 Pentium4 Core 2 Extreme X6800 只有 46.88GFLOPS。到目前为止，图形显示芯片拥有的新特征主要包括如下几点：

（1）在顶点级和像素级提供了灵活的可编程特性；

（2）在顶点级和像素级运算上都支持 32 位浮点运算，在某些阶段已经达到 64 位精度；

（3）支持绘制到纹理等功能，从而支持多遍的操作，这样避免了多次的 CPU 与 GPU 之间的数据交换；

（4）支持顶点纹理，从而在硬件上实现位移贴图；

（5）支持依赖纹理功能，以方便数据的索引访问，从而可以将纹理作为内存来使用。

目前，主流计算机中的处理器一般是中央处理器（CPU）和图形处理器（GPU），传统意义上 GPU 只负责图形渲染，而大部分的处理都交给了 CPU。随着图形硬件计算能力的提高和可编程特性的发展，人们将图形流水线的某些处理以及某些图形算法从 CPU 向 GPU 转移，基于 GPU 的图形算法复杂度也随之增加。除了计算机图形学本身的应用，其他领域的计算以及通用计算已成为 GPU 的应用之一，并成为研究热点。

基于 GPU 的通用计算[107,108]指的是利用图形卡来实现一般意义上的计算，而不单纯是绘制，借助于图形硬件可编程性和执行并行性的优势，可以充分利用计算机资源，分担 CPU 计算量，提高运行速度，常被用于计算集中型应用，如数值

运算[109]、粒子系统、流体仿真等。Joao[110]简要阐述了 GPU 的发展历史。Macedonia[111]认为 GPU 在 2003 年已经进入计算的主流，是通用计算的一个里程碑。2003 年 Li 等[112]采用 LBM（Lattice Boltzmann Method）来模拟流体和烟的效果。Kim 等[113]采用 GPU 来实施物理计算，仿真冰晶体生长过程。2005 年柳有权[114]借助 GPU 硬件加速方法实现基于物理的计算机动画。更多的图像硬件相关的内容可以从 GPU 的官方网站[115-119]中查询到。

### 2.5.1 硬件加速机制

随着当前计算机性能的不断提高，应用范围越来越广泛，不同的计算任务和计算需求都在快速增长，这就决定了处理器朝着通用化和专用化两个方向飞速发展。一方面，以 CPU 为代表的通用处理器是现代计算机的核心部件，经过多次器件换代的变迁，不仅集成度大大提高，性能和功能也大为改善，除了负责解释、执行指令和完成各种算术逻辑运算外，还控制并协调计算机各部分的执行。另一方面，处理器在特定领域应用的专用化程度也越来越高，例如在视频、图像和音频处理等领域，都出现了相应的专用处理器（如 VPU、GPU、APU 等）。在针对特定应用方面与通用处理器相比，专用处理器能够更加高效地满足特定的计算任务和需求。

在硬件架构上，CPU 与 GPU 具有本质区别。CPU 是一个串行处理器，同一时刻只能执行一条 CPU 指令，同一时刻只能处理一个数据，而 GPU 是一个并行处理器，一个 GPU 芯片中具有多个处理单元，在同一时刻可以执行多条指令以及处理多个数据。在“统一渲染架构”之前，GPU 中的处理器分为顶点渲染器和片段渲染器，在 NV80 系列 GPU 之后，“统一渲染架构”成为一种事实标准，每一个处理单元既可以作为顶点渲染器也可以作为片段渲染器。同时 GPU 的各处理单元之间已经实现了细颗粒度的线程间通信，可以由不同的处理单元组成数据处理流水线，从而大大提高数据处理的吞吐量。

GPU 和 CPU 在硬件设计上也具有很大不同。GPU 是一个专用型处理器，它的设计初衷就是为了快速地进行图形渲染，因此，GPU 在浮点运算上表现非常出众，在大规模数值计算方面，其速度远远优于 CPU；而 CPU 是一个通用型处理器，使用到的指令集大，能执行的任务种类多，在操作系统、系统软件、应用程序、通用计算、系统控制等领域有着优良的表现。相比于 CPU，图形硬件 GPU 有三个优点：

（1）计算速度比 CPU 的计算速度快，GPU 每秒钟可计算 1 万亿次，同样时间内 CPU 一般只计算几十亿次；

（2）显存的位宽比内存大，可一次计算的数据量大；

（3）程序的并行执行性，GPU 和 CPU 能够并行执行程序，GPU 内部有多个

渲染管道也可以并行执行。

GPU 在处理能力和存储器带宽上相对 CPU 有明显优势，在成本和功耗上也不需要付出太大代价，从而为大数据量的问题处理提供了新的解决方案。目前，越来越多的研究人员和商业组织开始利用 GPU 完成一些非图形绘制方面的计算，并开创了一个新的研究领域——基于 GPU 的通用计算（General-Purpose computation on GPU），其主要研究内容是如何利用 GPU 在图形处理之外的其他领域进行更为广泛的科学计算。目前已成功应用于代数计算、流体模拟、数据库应用、频谱分析等非图形应用领域，甚至包括智能信息处理系统和数据挖掘工具等商业化应用。

基于 GPU 的通用计算已成为近几年人们关注的一个研究热点。将 GPU 用于通用计算的主要目的是为了加速计算，加速的动力来自 GPU 在高性能计算方面所具有的优势：

（1）高效的并行性。这一功能主要是通过 GPU 多条绘制流水线的并行计算来体现的。在目前主流的 GPU 中，配置多达 16 个片段处理流水线，6 个顶点处理流水线。多条流水线可以在单一控制部件的集中控制下运行，也可以独立运行。通过多个渲染管道和 RGBA 四个颜色通道的同时计算，可以进行多个数据的同时计算。相对于并行机而言，GPU 提供的并行性在十分廉价的基础上，为很多适合于在 GPU 上进行处理的应用提供了一个很好的并行方案。

（2）高密集的运算。GPU 通常具有 128 位或 256 位的内存位宽，高于 CPU 上 32 位的位宽，这样整个计算的带宽大大提高，使得 GPU 相对于 CPU 来说，更适应传输大块的数据。同时，CPU 上的 Cache 一般只有 64KB，而现在的图形卡显存大多都在 256MB 以上，因此 GPU 在计算密集型应用方面具有很好的性能。

（3）超长图形流水线。GPU 图形流水线的设计以吞吐量的最大化为目标，因此 GPU 作为数据流并行处理机，减少了 GPU 与 CPU 的数据通信，在对大规模的数据流并行处理方面具有明显的优势。

上述这些优势使得 GPU 比 CPU 更适用于流处理计算，因此也被认为是一个 SIMD 的并行机或者流处理器，可以用于处理大规模数据集，使得应用得到加速。而相比之下，CPU 本质上是一个标量计算模型，而计算单元偏少，主要针对复杂控制和低延迟而非高带宽进行了若干优化。

到目前为止，有研究者将数据组织成顶点序列形成数据流，利用顶点程序来做基本的代数运算。但由于通常情况下，像素处理器的能力要比顶点处理器的能力强很多，所以大多数还是利用像素处理器来做通用计算。

### 2.5.2　硬件加速的编程实现方法

硬件加速分为两个应用方面，一种是针对渲染的加速，这种加速一般进行光

照、平滑、多重纹理合成等一些问题，渲染程序的执行结果直接输送到屏幕显示的帧缓冲器中，以进行屏幕显示。另一种是通用计算，这类应用常常是利用 GPU 针对大数据量处理的加速特性，将一些通用型的计算任务，由 CPU 移植到 GPU 上执行。这类程序的执行结果不能输送到屏幕，而是通过 FBO 技术保存到显存中，然后通过图形操作指令，将执行结果回传到内存中，从而进行进一步的计算。

GPU 的渲染流水线的主要任务是完成 3D 模型到图像的渲染（render）工作。渲染过程被分为几个可以并行处理的阶段，分别由 GPU 渲染流水线的不同单元进行处理。GPU 输入的模型是数据结构（或语言）定义的对三维物体的描述，包括几何、方向、物体表面材质以及光源所在位置等；而 GPU 输出的图像则是从观察点对 3D 场景观测到的二维图像。典型的 GPU 渲染过程包括：顶点生成、顶点处理、图元生成、图元处理、片元生成、片元处理、像素操作七个过程。在 GPU 渲染流水线的不同阶段，需要处理的对象分别是顶点（vertex）、几何图元（primitive）、片元（fragment）、像素（pixel）。

### 2.5.3 GPGPU 通用编程

GPGPU 在 Direct X 9 时代就已经初现雏形，随着 Direct X 10 时代的到来，GPU 与 CPU 都会相继加入相应的 GPGPU 技术。

从完整的 GPGPU 定义来讲，它不仅能够进行图形处理，而且能完成 CPU 的运算工作，更适合高性能计算，并能使用更高级别的编程语言，在性能和通用性上更加强大。从狭义的 GPGPU 应用来说，GPGPU 就是功能强化的 GPU，GPU 的优势也正是 GPGPU 的优势，弥补了 CPU 浮点运算能力的严重不足。

GPGPU 计算通常采用 CPU + GPU 异构模式，由 CPU 负责执行复杂逻辑处理和事务管理等不适合数据并行的计算，由 GPU 负责计算密集型的大规模数据并行计算。但是，传统的 GPGPU 受硬件可编程性和开发方式的制约，应用领域受到了限制，开发难度也很大。

最初的程序员在编写着色程序时需要使用汇编语言，该语言难度大、效率低，如今 GPU 的高级着色语言能够极大地提高程序员的编程效率。微软公司的 HLSL、OpenGL 的 GLSL、斯坦福大学的 RTSL，以及 NVIDIA 公司的 Cg 等高级着色语言都能够隐藏底层硬件的技术细节，提高 GPU 的开发效率。

最早的 GPGPU 开发直接使用图形学 API 编程，它要求编程人员将数据打包成纹理，将计算任务映射为对纹理的渲染过程，用汇编或者高级着色器语言（如 GLSL、Cg、HLSL）编写 shader 程序，然后通过图形学 API（Direct 3D、OpenGL）执行。这种方式要求编程人员不仅要熟悉自己需要实现的计算和并行算法，还要对图形学硬件和编程接口有深入的了解。在 GPGPU 计算中，从 GPU 中返回到系

统内存中数据，首先需要分配和初始化在内存中的空间，然后调用 glGetTexImage 函数，将纹理数据返回到该 Tex 内存中，并以实际大小存入 Vector 中。

2007 年 2 月，NVIDIA 公司正式发布了 CUDA 架构（Compute Unified Device Architecture，统一计算设备架构），这也是 NVIDIA 公司确定的 GPGPU 产品的正式名称。CUDA 是 GPGPU 产品的一个新的基础架构，也是世界上第一个针对 GPU 的 C 语言开发环境的 GPGPU 产品，它采用 C 语言作为编程语言提供大量的高性能计算指令开发能力，使开发者能够在 GPU 的强大计算能力的基础上建立起一种效率更高的密集数据计算解决方案，并且提供了硬件的直接访问接口，而不必像传统方式一样必须依赖图形 API 接口来实现 GPU 的访问。

## 2.6　本章小结

本章针对计算机角色群组仿真中所涉及的关键技术进行了分析和总结，分别包括角色群组仿真过程中的角色驱动技术、角色模型变形技术、角色快速绘制技术以及图形硬件加速技术。对于这些技术，分析了近十年以来的发展情况，并对相关论文进行概要的分析。接下来的几章中，将分别针对各种相关技术进行详细分析，并讲述和总结作者的研究成果。

# 第 3 章　二维运动角色变形技术研究

计算机生成场景是对现实世界的一种模拟，而人物角色是现实世界中的一个重要组成部分，因此对人物角色或动物角色的仿真是计算机图形学领域的一个重要内容。各类运动角色的仿真大多是使用模型变形技术实现，模型变形技术将静态的模型变形为所需要的形状、姿态或行为。

在计算机虚拟角色仿真中，三维与二维模型都很常用，它们都能进行较真实的动作模拟，其不同点主要体现在模型特点和应用领域两个方面。图形形式的三维角色具有变形结果精度高、细节明显、表现力强等特点，但三维角色建模过程与变形计算复杂、控制复杂，变形结果渲染对运行系统的要求高。图像形式的二维角色，具有获取方便、变形计算量小、渲染对设备需求低的特点，尤其重要的是二维角色拥有庞大的现有资源库，而且制作门槛低，可以快速制作。因此，虽然三维角色具有二维角色不可比拟的表现力，但二维角色仍然具有强大的生命力。

目前现有的二维角色变形方法中，较新的实现方法是 2005 年 Igarashi[15] 和 2006 年 Schaefer[73]、Weng[74] 等人使用几何网格覆盖人物或动物角色图像，以数学方法计算网格顶点的位置，使用图形的方法解决图像的变形问题。但这类变形方法对于大规模运动角色的仿真应用仍存在以下 3 方面的不足：

（1）网格顶点坐标都是通过数值方法求解，不可避免地存在算法复杂、计算时间长等问题，造成所能支持的变形顶点数量局限在 300 个左右；

（2）这些方法都是全局变形，均有不同程度的变形结果失真，如不平滑尖角现象和不合理的变形结果等，这种现象的原因是由于全局变形中每一个位置对整体的贡献率相同，面积小的位置容易失真；

（3）由于使用数值方法求解网格顶点，使得算法复杂度高，无法应用图形硬件加速角色变形计算。

本章针对 2D 人物图像或动物图像的变形问题，研究变形模型更简洁、运行更快速的 2D 角色变形算法，从而支持大量 2D 角色同步变形。针对此目的，本章提出一种可以用于二维角色图像变形的自适应网格。自适应网格是指可以自动适应角色姿态变化的网格，该网格由曲线控制的三角带和长度补足矩形组成，可以自动实现以下三个方面的调节：

（1）根据当前的角色姿态计算网格控制曲线的参数，改变网格形状，从而实现覆盖区域的角色图像变形；

（2）根据关节点旋转角调整控制曲线间距，保持网格覆盖区域的总面积不变；

（3）每个自适应网格对应一个角色关节点，互相连接的关节点网格可以实现无缝接合。

基于自适应网格的二维角色变形方法变形操作简单，计算量小，可以实现大量角色的实时变形。在变形结果真实感方面，通过控制曲线的导数连续性，保证变形区域的弯曲平滑性。通过变形区域的面积保持，有效避免尖角等失真现象。在变形计算实时性方面，通过影响因素局部化以及 GPGPU 图形硬件通用计算加速变形速度。该方法适用于人物和动物等二维角色的变形，实验结果证明该方法是可行的，能够有效避免角色变形结果的不合理和失真问题，并提高角色变形速度。

本章的其他部分将包含如下内容：

（1）介绍关于二维角色的图像变形技术及其现状；

（2）介绍本章提出的基于自适应网格的变形方法，包括简化骨骼信息获取、自适应网格的构建及其面积保持方法、硬件加速方法以及网格间的无缝连接验证；

（3）对提出的方法进行验证，与他人的变形结果进行比较，并进行运行时间分析。

## 3.1　相关工作

随着计算机图形学技术的发展，模型建模技术得到很大的提高，有很多种软件可以对所需的物体建立模型，但这些模型大多是针对所描述物体的一个静止状态建立的，只能表示物体的一个形态。物体变形技术通过改变模型的顶点位置或其他属性，使模型展现出不同的形态，从而丰富虚拟世界的多样性。为实现物体变形，研究者们提供了许多种支持变形的模型以及针对这些模型的变形方法。

虚拟角色的运动主要靠角色模型的变形来实现，模型变形技术根据应用对象主要分为二维图像变形和三维模型变形。传统的图像变形主要有基于 Morphing、基于 FFD 和基于骨骼的变形方法。Collins[34] 介绍了各种角色动画制作与模型变形中使用的模型及其变形原理，Lewis[38] 将各类变形方法统一用 Pose 空间变形解释，并使用 Morphing 形状插值和径向基函数进行变形计算。

Morphing 变形是指将一给定的源数字图像和几何对象 S 光滑连续地变换到目标数字图像或几何对象 T，这种光滑过渡中，中间帧既应该具有 S 的特征，也需要具有 T 的特征。S 和 T 的拓扑既可以相同也可以不同。源处理对象和目标处理对象之间的特征对应关系，可以由动画师指定，也可以由动画系统自动计算求

得。Morphing 技术大体可以分为二维图像 Morphing 技术、二维形状 Morphing 技术和三维图形 Morphing 技术。

二维图像 Morphing 是指把一幅数字图像以一种自然流畅的、戏剧性的方式变换到另一幅数字图像，对于二维图像 Morphing 的方法主要有：

（1）场景编辑法，也就是通过一定的景物遮挡来实现；

（2）停住运动动画法；

（3）交融技术，也称淡入淡出技术；

（4）二维粒子系统技术，将图像解体然后重组。

在二维形状动画中，经常会给定一个初始和最终的形状，求从初始形状平滑过渡到最终形状的中间形状。给定的初始形状和最终形状为关键帧形状，而中间形状的求取过程称为二维形状 Morphing。

2003 年偶春生[42]通过两幅图像的边界点对应，实现角色图像的 Morphing 过渡，2008 年 Guo[44]通过自动计算形状拓扑并匹配轮廓顶点实现图像 Morphing，并保持形状边长及其局部、全局和边界属性。Morphing 变形一般算法简单，但必须需要两幅图像才能插值出中间图像，导致其使用范围受到很大的限制。

FFD 变形方法将整幅图像看作一个空间，通过空间的变形实现图像内容的变形。MacCracken[49]使用点阵网格划分变形空间进行 FFD 变形，2000 年 Chua[52]提出一种硬件加速的 FFD 变形方法。FFD 方法只需一幅图像即可进行变形，通过空间变形实现图像变形，变形原理简单，与角色的拓扑结构无关，但 FFD 变形结果容易扭曲过度，而且变形控制复杂，获得所需的变形结果很困难。

基于骨骼的图像变形方法是根据角色骨骼结构将图像划分为若干个区域，每个区域使用 FFD 方法实现变形。Capell[57]根据变形模型的骨骼结构把图像划分为若干子区域进行变形，Yan[61]使用三角辅助网格覆盖变形角色图像，通过边缘控制实现基于骨骼的图像变形，Stalpers[14]用多边形网格模拟骨骼进行几何模型变形，Yang[68]使用曲线控制三角形网格实现变形。骨骼模型符合现实世界中动物的肢体结构，因此骨骼变形方法容易获得接近真实的变形结果，而且局部变形较易处理大模型的变形，但骨骼模型的建立过程复杂，各段骨骼对顶点的影响权值配置复杂，另外在各个子区域的连接处，容易出现不平滑等失真现象。

近几年来出现了使用几何网格覆盖角色图像，通过几何网格的变形获得图像变形结果的方法，如 2005 年 Igarashi[15]使用三角网格覆盖作为辅助网格，2006 年，Yoshioka[75]使用间隔提取像素点作为变形辅助网格，以控制顶点作为约束条件，使用最小二乘法求解网格自由顶点，实现虚拟角色的图像变形，Schaefer[54]将变形中的每一个影响因素表达为方程式，使用最优化求解方程组，获得自由顶点坐标和图像变形结果实现变形，Weng[74]将图像角色网格化，把变形看作优化问题来求解图像变形结果。这些方法变形控制简单，而且经过复杂的数值运算，

可以获得较好的变形结果。但这些方法存在的一个问题是全局变形计算使得运行时计算量大，任一顶点的位置改变都将引起整体重新计算，大模型的变形实时性难以保障，另一个问题是这些方法可能会出现不真实或不合理的变形结果。

随着图形硬件计算能力的提高和可编程特性的发展，人们将图形流水线的某些处理以及某些图形算法从 CPU 向 GPU 转移，基于 GPU 的图形算法复杂度也随之增加。除了计算机图形学本身的应用，其他领域的计算以及 GPGPU 通用计算[107,108]都已成为 GPU 的应用之一，并成为研究热点。

GPGPU 是继 GPU 出现之后，对图形硬件功能的又一大扩展，它利用图形硬件的多通道、高位宽等特点进行高速通用计算，通过离屏渲染等技术，保存计算结果并将其返回到内存。基于 GPU 的通用计算不单纯是利用图形卡来实现绘制，而是借助于图形硬件可编程性和执行并行性的优势，实现一般意义上的计算，分担 CPU 的计算量。但图形硬件的设计特点，使 GPGPU 通用计算并不能执行所有 CPU 程序，图形硬件可高速执行控制结构简单的程序，其程序长度与功能都有很大的限制。对于以往的覆盖网格变形方法，由于其算法的复杂性超出了图形硬件所能执行的能力，无法在图形硬件上实现，这在一定程度上也限制了算法的优化和改进空间。

相比于以往的图像变形方法，本文提出的方法主要不同之处在于：

（1）采用简化骨骼信息建立自适应变形网格，只需对变形部分标定简化骨骼框架，解决以往骨骼模型的建立过程复杂的问题；

（2）利用自适应网格的平滑特性，实现关节点连接处的平滑变形，避免尖角等问题；

（3）利用自适应网格的总面积保持特性，实现变形过程中的面积保持，有效减小变形引起的形状面积失真；

（4）利用自适应网格的局部运算特性和网格顶点的规则分布特性，实现图像变形算法的硬件加速。

## 3.2　基于自适应网格的二维角色变形

合理逼真的变形结果和快速高效的变形计算是变形系统追求的核心目标。在变形过程中，合理逼真的变形结果需要通过减小变形失真来实现，如何减小变形失真与变形方法的设计紧密相关。对于快速的变形计算不仅需要通过设计良好的变形算法来获得，而且需通过有效的加速方法来提高算法的执行速度。

覆盖网格的变形方法，是目前最新的变形方法，这类方法通过图形的方法来解决图像的问题，使用几何图形的计算可控性实现方便的角色图像变形。在这些基于覆盖网格的变形方法中，变形造成尖角或不平滑等失真问题是由于全局变形

方法的特点决定的，全局变形的每一个位置对整体的贡献率是相同的，根据面积计算贡献率，如四肢部分的面积较小而躯干部分面积较大，躯干部分控制点的微小改动就会引起四肢的大幅度变形，而真实的角色变形中，各关节的贡献率是不同的。

覆盖网格变形算法的复杂性主要是由其算法实现原理决定的，这些方法在覆盖网格中选择控制顶点，以控制顶点的位置为条件，使用数值方法——最小二乘法求解自由顶点的位置坐标。这相当于一个逆向求解的过程，通过不断迭代更新自由顶点的位置，直到它们适应控制顶点的位置为止。这种数值运算造成变形网格的顶点计算难度很大。

基于上述的分析，需要对变形角色图像根据骨骼信息划分变形区域，通过隔离各关节点之间的联系，将全局变形局部化，以解决全局变形方法中关节贡献率全部相同的问题，避免关节点周围的尖角等失真，实现较逼真的变形结果。同时，使用基于骨骼的表达方式定义角色图像拓扑，构建依赖于骨骼结构的较规则的覆盖网格，根据关节点计算覆盖网格顶点的位置，将网格求解的逆向过程转化为正向过程，简化其计算方法，并研究其硬件加速方法。

本节提出了自适应网格，该网格以双曲线作为每个关节点变形区域的边界控制线，相邻的关节点区域的边界线导数连续，使连续区域可以平滑衔接。通过改变两侧边界控制线的形状实现区域变形，通过曲线的平滑特性避免以往骨骼变形中关节连接处的尖角问题，降低不平滑失真。

### 3.2.1 自适应网格的原理

人类或其他脊椎动物的运动均为骨骼运动，使用基于骨骼的模型表达更为直观，使用骨骼模型进行角色图像的变形也更为合理。现实中动物角色的组成是以骨骼为支撑，在骨骼外包裹肌肉，角色的动作过程中骨骼绕关节点旋转，而骨骼上的肌肉则随骨骼的旋转而发生挤压或拉伸——外侧肌肉被拉伸而内侧肌肉被挤压，旋转的角度变化越大，则挤压或拉伸的形变程度就越大，而另一方面，在同一旋转角度下，越靠近关节点附近的位置肌肉的形变越大，越位于每段骨骼中间部分——远离关节点的位置形变越小。

根据关节点划分变形区域，以形变最大的关节点作为变形区域中心，以变形贡献最小的位置——骨骼中间点作为变形区域的分割点划分变形区域，将整个角色的全局变形分割为局部变形。在骨骼角色的变形过程中，关节点周围的区域变形程度最大，角色的关节点变形主要发生在其自适应网格内部。肌肉体的弹性，使动物表皮的外边缘较光滑，在靠近关节点的位置形成较圆滑的曲线，而在远离关节点的位置形变很小，近似于拉直状态的直线状态，该属性比较接近双曲线的形状属性。

将以关节点为中心的变形区域边界使用双曲线进行表示，表示效果如图 3-1 自适应网格构建原理图所示。图 3-1（左）是该变形区域在两段骨骼成平角时的外围形状，图中的虚线表示变形区域的中心线，在平角状态下中心线即为该区域的骨骼线，$W_B$ 是该区域的宽度，这个宽度需要稍大于角色图像在该区域的宽度，以此保证网格可以覆盖角色。图 3-1（右）为关节点两侧的骨骼成某角度时，双曲线表示的变形区域的外围形状，图中的虚线是该变形区域的中心线，中心线在变形过程中将保持长度不变，*OM* 和 *ON* 是关节点两侧的两段骨骼，$\alpha$ 是两段骨骼在该姿态下的夹角，从图中可以看出，在靠近关节点的附近位置，变形区域的内侧边缘线变短而外侧边缘线变长，同时两侧双曲线边界距离由拉直状态下的 $W_B$ 变宽为 *iX*，这恰好符合受挤压的变形理论。

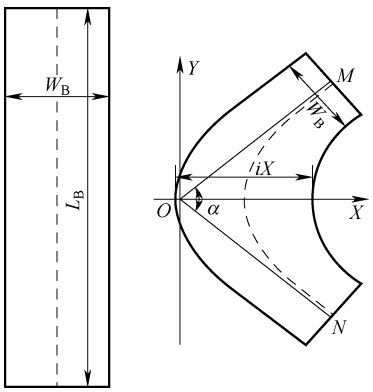


图 3-1　自适应网格构建原理图

综上所述，使用双曲线进行边界控制的自适应网格，可以模拟骨骼周围附带肌肉等软组织受挤压所形成的变形结果，能够有效地实现其变形仿真。

### 3.2.2　自适应网格的关节点旋转角计算

在自适应网格的构建和绘制过程中，关节点旋转角都是一个重要的参数，为了便于问题的描述，在此对关节点旋转角给出准确的定义。在骨骼角色中，每个非首尾关节点都有前后连接的两段骨骼，从上段骨骼沿顺时针旋转至下段骨骼形成一个旋转角，该旋转角称为该关节点旋转角。如图 3-1（右）中，关节点两侧的骨骼为 *OM* 和 *ON*，两骨骼的夹角为 $\alpha$，当角色图像的渲染顺序为 $M \rightarrow O \rightarrow N$，即 *OM* 为上段骨骼，则关节点 *O* 的旋转角为由 *OM* 顺时针旋转到 *ON* 的角度，即 $\alpha$。

关节点旋转角表示该关节点前后的渲染方向的变化，也表示该关节点前后局部坐标系与全局坐标系的转换关系。角色的姿态变化是通过各段骨骼围绕其前端关节点旋转完成的，当模型发生变形，则对应关节点旋转角会同时发生变化，反之，关节点的旋转角也决定了该关节点的当前姿态——该原理将用于本文第 4 章的动作驱动方法。

一条骨骼链上的关节点旋转角的具体计算方法如图 3-2 所示，图中 *ABCD* 为一条四关节点的骨骼链。在这四个关节点中，首尾两个关节点 *AD* 只有一侧骨骼没有旋转角，另外两个关节点 *BC*，在每个关节点上以当前关节点为中心，从其上一关节点沿顺时针方向旋转至其下一关节点的角度，即为该关节点的旋转角，

如关节点 $B$ 的 $\angle\beta$ 和关节点 $C$ 的 $\angle\gamma$。

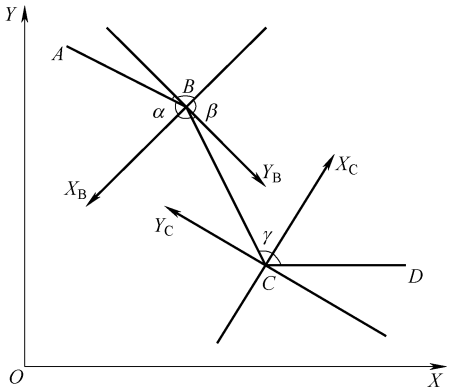



图 3-2 关节点旋转角计算原理图

骨骼链中的一个关节点的旋转角可以用式（3-1）计算，即

$$\beta=\begin{cases}\alpha & ,flag\geqslant 0\\ 2\pi-\alpha & ,flag<0\end{cases} \tag{3-1}$$

式中，$\alpha$ 可以用余弦公式求得；$flag$ 为关节点旋转角的标记符，可用式（3-2）计算获得；$flag\geqslant 0$ 表示旋转角小于平角；$flag<0$ 表示旋转角大于平角。

$$flag=(y_A-y_B)(x_C-x_B)-(y_C-y_B)(x_A-x_B) \tag{3-2}$$

对于图 3-2 中的关节点 $B$，其局部坐标系以 $B$ 点为原点，以 $BA$ 和 $BC$ 两射线的角平分线为 $X$ 轴，渲染次序为 $A\rightarrow B\rightarrow C$，其中关节点 $B$ 的旋转角为 $\beta=2\pi-\alpha$。而对于关节点 $C$，则局部坐标系以 $C$ 点为原点，关节点 $C$ 的旋转角即为 $\gamma$。

在变形过程中，关节点的旋转角与下一关节点的位置息息相关，当骨骼链中的某一关节点旋转角发生变化时，同骨骼链的所有关节点都需要重新计算其位置，并根据新的局部坐标与全局坐标的变换关系，重新计算其变形模型的网格顶点坐标。如在图 3-2 中，关节点 $C$ 的位置变换引起 $B$ 的旋转角发生变化，而关节点 $B$ 的旋转角发生变化，也引起 $CD$ 两关节点的位置发生变化。以关节点 $B$ 的旋转角变化为例，关节点 $C$ 的位置可用式（3-3）计算得出，其中 $\theta_{BC}=\theta_{AB}-\beta$ 是射线 $BC$ 的方向角。

$$\begin{cases}x_C=x_B+L_{BC}\cos(\theta_{BC})\\ y_C=x_B+L_{BC}\sin(\theta_{BC})\end{cases} \tag{3-3}$$

### 3.2.3 自适应网格的构建

自适应网格因为其可自动调节的特性，能够支持角色图像的变形，分散图像变形过程中的失真，提高变形结果的逼真度，自适应网格将变形区域划分为三角

带，使变形失真均匀分散到整个变形区域，并通过调整网格构建参数，使网格总面积在变形过程中保持不变，进一步降低变形失真。

本章提出的自适应变形网格，是针对骨骼变形角色的单个关节点设计，网格以关节点为中心，覆盖关节点的周围区域。自适应网格的构建依据是角色的简化骨骼信息，包括角色的变形关节点位置及其连接关系。自适应网格的构建过程可以描述为

（1）根据关节点所在区域的大小信息计算网格覆盖区域；

（2）根据该关节点的旋转角，在局部坐标系中计算网格顶点的局部坐标，获得该关节点的局部坐标系网格；

（3）根据局部坐标和全局坐标的转换关系，将各关节点的网格变换到全局坐标系，并连接为一个整体。

角色姿态的改变会引起关节点旋转角变化，从而使自适应网格中的三角带形状的变化，整个角色的变形可以通过改变各关节点的自适应网格形状来实现，任意一个关节点对应的自适应网格发生变化，都会造成其同一关节链关节点的局部与全局坐标的转换关系发生变化。

单个关节点的自适应网格所在的局部坐标系，以当前关节点位置为坐标原点，以关节点两侧骨骼的角平分线作为坐标系 $X$ 轴。自适应网格由均匀分布的三角带网格组成，三角带网格在局部坐标系中关于 $X$ 轴上下对称。

自适应网格的三角带网格均匀分布在模型中心线上，三角带顶点位于两侧的边界双曲线上，曲线的参数由当前关节点相连骨骼的长度宽度以及夹角角度决定。其构建原理如图 3-3 所示，为简化计算过程，加快计算速度，根据双曲线关于 $X$ 轴上下对称的性质，只计算 $X$ 轴以上的部分网格顶点，然后通过 $X$ 轴对称获得所有的变形网格顶点。

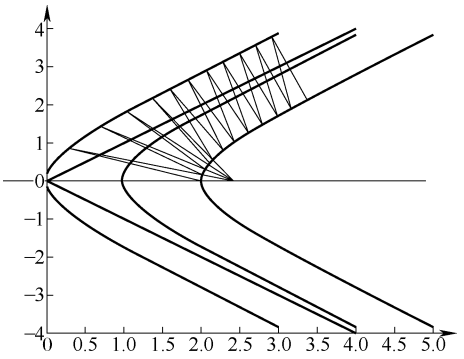



图 3-3　自适应网格构建原理图

**1. 自适应网格的中心线**

计算自适应网格的第一步是计算网格的边界曲线。两侧边界双曲线通过中心双曲线向两侧平移获得，中心双曲线以两侧骨骼作为渐进线，其表示方程为式（3-4）。

$$\frac{X^2}{r^2}-\frac{Y^2}{(kr)^2}=1 \tag{3-4}$$

式中，$r$ 为硬度参数，可以由用户用来调节肌肉的柔软度，$k$ 为夹角参数，正比于关节点两连接的骨骼之间的夹角大小，在变形过程中，该参数响应用户的变形

操作，改变此参数即可改变网格的最终形状。

**2. 自适应网格的边界双曲线**

两侧边界双曲线由中心双曲线在 $X$ 轴方向平移得到，定义该关节点的变形子区域。内外侧双曲线的方程为式（3-5）。

$$\frac{(X \pm iX)^2}{r^2} - \frac{Y^2}{(kr)^2} = 1 \tag{3-5}$$

该方程中 $iX$ 为两侧双曲线与中心双曲线的 $X$ 轴方向平移量，变形中根据关节点的旋转角计算，它也是实现网格总面积保持的重要参数，其计算方法在面积保持部分将详细介绍。

**3. 均匀分布的中心线分割点**

为分散图像变形过程中的失真，提高变形结果的逼真度，双曲线部分使用均匀分布的三角带来覆盖。均匀分割模型中心线，得到三角形与中心线的交点坐标。中心双曲线的长度可用式（3-6）计算。

$$\int_L f(X,Y)\,\mathrm{d}l = \int_0^{y\mathrm{Max}} \left(\sqrt{(\partial f(X,Y)/\partial Y)^2 + 1}\right)\mathrm{d}Y \tag{3-6}$$

式中，$f(X,Y)$ 为中心双曲线方程；$y\mathrm{Max}$ 为自适应网格所覆盖范围的最大 $Y$ 值。根据中心线的长度，获得中心线的分割点，这些分割点将均匀地分布在中心线上。

**4. 边界双曲线上的网格顶点**

根据中心线上的分割点，将分割点映射到两侧的边界双曲线上。以通过分割点的中心线垂线作为分割映射线，与边界线相交获得两侧边界上的分割点。当垂线无法与内侧边界相交时，使用过焦点的直线作为映射线，分割映射线的斜率可用式（3-7）求得。

$$k = \min\left(\frac{-1}{f'(x,y)}, \frac{yDiv_i - 0}{xDiv_i - xFocus}\right) \tag{3-7}$$

式中，$f(x,y)$ 为中心双曲线方程；$(xDiv_i, yDiv_i)$ 为分割点坐标；$(xFocus, 0)$ 为内侧曲线焦点。

**5. 两侧骨骼长度不同时的补足矩形**

关节点两侧的骨骼长度往往不相同，此时，需要添加一个骨骼长度调节矩形来补齐其长度差异，该矩形称为长度补足矩形，补齐位置为弯曲变化率最小的骨骼中间部分区域。长度补足矩形的宽度为较长侧的骨骼标定宽度，长度为两侧骨骼的骨骼半长度之差。对于只有一侧骨骼的关节点，可以看作另一侧的骨骼长度为零，自适应网格退化为一个长度补足矩形。

至此针对单个关节点的自适应网格建立完毕，关节点的变形子区域被划分为一个以关节点为中心两侧对称的双曲线区域和一个补齐骨骼间长度差异的矩形区域。

### 3.2.4　自适应网格的面积保持

面积保持和体积保持是目前变形方法中减小模型形状整体失真的有效手段，在二维角色图像变形过程中一般使用面积保持。图像变形的面积保持特性可以有效地避免一些折叠、收缩类的不合理变形结果，以降低角色形状的失真度。

面积保持又分为全局面积保持和局部面积保持，全局面积保持是变形过程中整幅图像或整个角色的总面积保持不变，局部面积保持在变形过程中保证整幅图像总面积保持不变的同时，还要保持各个子区域面积不变。

在目前的图像变形方法中，面积保持已经成为一种必须实现的算法特性。自适应网格作为一种应用于二维角色图像变形的支持网格，同样也需要实现面积保持的特性。在以往的覆盖网格图像变形方法中，面积保持是通过放缩三角面片的大小来实现的。例如在 Igarashi[15] 方法中通过分别计算每个对应三角形的面积变化系数，进行放缩变换，从而得到面积保持的变形结果。值得注意的是，这种放缩需要很多步的计算，放缩之后还需要对三角形进行拼接，这是一个很繁琐的计算过程。

相比于这种繁琐的面积保持计算方法，本章提出的自适应网格的面积保持，由以往的图像变形后调节，改进为变形前预测调节，从而可以更方便地实现变形过程中的面积保持。由于自适应网格是针对单个关节点进行构建的，各个关节点的变形计算是互相独立的，在变形过程中，只需要保持每个自适应网格的面积，即可实现局部面积保持特性。以下是自适应网格在变形过程中实现面积保持的原理及其计算方法。

自适应网格由三角带网格和长度补足矩形组成。在角色动作变形时长度补足矩形形状不变，其面积在变形中也保持不变。三角带网格负责实现角色的形状改变，当角色姿态变化时，关节点旋转角改变，自适应网格的形状将随之改变，各个三角形面积也会发生变化。

由自适应网格的构建过程可知，在变形过程中，长度调节矩形的面积不会改变，要保持网格总面积不变，只需保持变形前后三角带网格的总面积不变即可。在自适应网格变形中，通过调节两侧边缘曲线之间的 $X$ 距离，可以保持三角带网格的总面积不变，从而实现自适应网格的总面积保持，亦即实现角色变形的局部面积保持。图 3-4 所示为面积保持原理图，其实现过程如下：

**1. 证明中心三角带总面积与矩形 *abcd* 面积相同**

图 3-4a 中的三条曲线分别是自适应网格的中心双曲线和两侧边界双曲线。两侧双曲线由中心双曲线在 $X$ 轴方向上平移得到，它们的形状参数相同，因此区域 *amb* 与 *dnc* 的面积相同。根据双曲线的性质，当渐近线表示的骨骼长度几倍于宽度时，端点处曲线与渐近线距离很近，所以 *aef*、*dgf*、*bpk* 和 *chk* 四个区域面积近

似相同。综上可知，自适应网格的三角带部分面积与矩形 *abcd* 的面积相同。等效面积替换的过程如图 3-4a 所示。

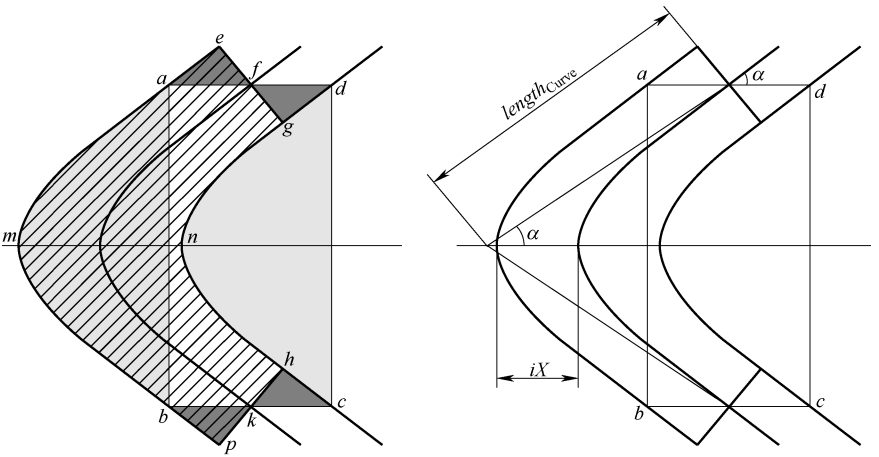



图 3-4　自适应网格的面积保持原理图

a）等效面积替换示意图　b）面积计算示意图

**2. 计算矩形 *abcd* 面积**

根据双曲线渐近线的性质，图 3-4b 中矩形 *abcd* 的宽 $width = 2iX$，高 $height = 2length_{Curve}\sin\alpha$，而矩形 *abcd* 的面积 $S_{abcd}$ 计算方法见式（3-8）。

$$\begin{aligned} S_{abcd} &= height \cdot width \\ &= 4length_{Curve}\sin\alpha iX \end{aligned} \tag{3-8}$$

式中，$length_{Curve}$ 为曲线式骨骼模型中心三角带覆盖的骨骼长度；$\alpha$ 为关节点两侧骨骼半夹角即渐近线倾斜角；$iX$ 为两侧曲线相对于中心线的 $X$ 平移量。面积计算示意图如图 3-4b 所示。

**3. 计算两侧曲线的 *X* 平移量 *iX***

角色的骨骼长度在变形中保持不变，由式（3-8）可知，要使变形模型中心三角带总面积保持不变，只需使 $\sin\alpha iX$ 在变形中保持不变即可。$iX$ 计算公式可以表示为式（3-9），其中 $Bonewidth_0$ 为初始状态的骨骼宽度。

$$iX = Bonewidth_0 / \sin\alpha / 2 \tag{3-9}$$

通过等价三角形置换，以及计算两侧边界双曲线的平移距离 $iX$，自适应网格的三角带面积即矩形 *abcd* 的面积可以用式（3-10）计算获得，此面积不随变形中骨骼的夹角变化而变化，因此在变形中可以保持面积不变。

$$\begin{aligned} S_{abcd} &= 4length_{Curve}\sin\alpha iX \\ &= 2length_{Curve}Bonewidth_0 \end{aligned} \tag{3-10}$$

## 3.3　二维角色变形的硬件加速

### 3.3.1　自适应网格的图形硬件加速

对于计算机虚拟角色的仿真，人们不仅需要形象的真实感，也需要运动的真实感，同时对实时性的要求使得仿真过程必须对算法本身进行优化和改进，一方面提出新的算法，另一方面也要求提高现有平台的计算能力。在计算机图形处理中，除了内存和 CPU 外，图形硬件加速也是一种重要的加速手段。

在大规模运动角色仿真应用中，对于当前的二维角色覆盖网格变形和三维角色细节保持变形，其算法复杂度都很高，而且需要执行拓扑结构的重新计算，超出了图形硬件所能承受的范围。本章提出的自适应网格，由于其局部运算特性以及三角带网格顶点的规则分布特性，使得自适应网格十分有利于在 GPU 上进行计算。使用 GPGPU 技术加速计算自适应网格的网格顶点坐标，可以减小 CPU 计算压力，以提供更流畅的变形渲染。采用 GPGPU 加速后，CPU 负责用户交互等操作，GPU 负责自适应网格计算，充分利用 CPU/GPU 并行计算能力加速变形渲染。

**1. GPGPU 加速实现方法**

在图形加速硬件中有两种图形渲染器——顶点渲染器和片段渲染器，这两种渲染器中均可进行用户编程并执行 GPGPU 程序。GPU 图形硬件渲染的流水线如图 3-5 所示，顶点渲染器中的程序，对于每个传入的顶点执行一次，执行的次数与传入的顶点数有关。顶点渲染程序的处理结果输出至片段渲染器进行处理，片段渲染器中的程序，对于每个渲染至帧缓冲区的片段执行一次，执行的次数与最终的输出片段数有关。

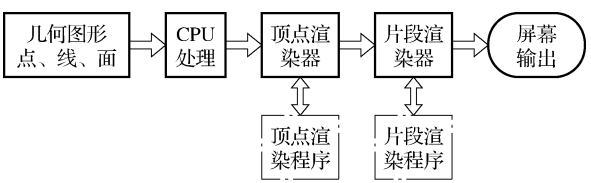



图 3-5　图形硬件渲染流水线示意图

GPGPU 通用计算，可以有两种实现方法，分别是二次渲染方法（Second Rendering）和一次渲染方法（Single Rendering）。在每一帧变形渲染过程中，二次渲染方法需要执行两次渲染循环，第一次渲染循环主要执行计算功能，并返回计算结果，第二次渲染循环执行实际屏幕绘制。而一次渲染方法，在一个渲染过程中同时完成计算和屏幕绘制两种功能。二次渲染方法，一般只使用片段渲染器

进行求解，渲染中的每一个片段对应一个计算数据点，用于纯计算应用，如矩阵计算，粒子求解等；而一次渲染方法因为需要同时完成计算和屏幕绘制两种功能，不能切换渲染目标，所以一次渲染方法只能完成简单的几何图形属性计算。对于简单的几何图形属性计算，可以选择二次渲染方法，也可以使用一次渲染方法来实现。

在 GPGPU 加速的自适应网格顶点计算应用中，传统 CPU 计算、GPGPU 二次渲染计算和 GPU 一次渲染计算三种方法之间的区别见表 3-1。在二次渲染方法中，整个计算过程分为三部分：初始化部分、计算部分和重置部分。在 GPU 一次渲染计算中，整个计算过程和 CPU 计算一样，只需要计算部分，这样在数据传输量较大的计算中可以保持运算速度的优势。

**表 3-1　自适应网格的三种计算方法区别**

| | 传统 CPU | GPU 二次渲染法 | GPU 一次渲染法 |
|---|---|---|---|
| 每帧设置/重置渲染目标 | 否 | 是 | 否 |
| 使用 FBO 对象 | 否 | 是 | 否 |
| 主要计算单元 | CPU | 片段渲染器 | 顶点渲染器 |
| 主要计算单元速度 | 慢 | 快 | 快 |
| 主存与显存的数据交换 | 无 | 数据上传与下载 | 数据上传 |

**2. 二维角色变形的二次渲染加速方法**

在传统的 GPU 渲染流水线中，每次渲染运算的最终结束点就是帧缓冲区，然后实时地输出到屏幕显示出来。渲染到纹理技术[120]允许将一个 FBO（Frame Buffer Object）离屏缓冲区作为渲染运算的目标，而不会直接输出到屏幕，渲染计算的结果保存到 FBO 绑定的纹理中，然后可以通过纹理操作，进行显存和内存之间的数据交换。GPGPU 应用的计算结果，可以通过该方法返回计算结果。

二次渲染方法在每一帧中需要两个渲染循环，第一次渲染循环为离屏渲染，计算三角带顶点坐标并保存至 FBO 离屏缓冲区，通过渲染至纹理技术将三角带网格顶点坐标返回到内存，需同时使用顶点渲染器和片段渲染器。

GPU 二次渲染加速方法的算法执行过程可以描述为

步骤 1：由 CPU 根据用户交互操作计算关节点的新位置坐标，将更新后的关节点坐标数据上传到显存中。

步骤 2：设置渲染目标为 FBO 离屏缓冲区，由 CPU 通过渲染通道渲染一个矩形，矩形的每一行片段对应一个关节点，每一个片段对应一个网格顶点。

步骤 3：在 GPU 顶点渲染器中计算各关节点的自适应网格参数及其局部坐标系的坐标转换矩阵等。

步骤 4：在片段渲染器中计算关节点的自适应网格顶点的坐标，并转换为全局坐标。计算结果保存至 FBO 的离屏缓冲区。

步骤 5：将 FBO 离屏缓冲区中的数据取回内存，设置渲染目标为屏幕帧缓冲区，根据返回的计算结果，按三角带类型实际渲染到屏幕帧缓冲区，并输出至屏幕。

**3. 二维角色变形的一次渲染加速方法**

在图像变形的硬件加速应用中，GPU 执行的是几何图形的属性，加速过程中对变形模型的顶点位置坐标、纹理坐标进行计算，可以使用一次渲染方法实现加速计算过程。

对于顶点位置计算的一次渲染方法，只需在顶点渲染器中进行计算，每一个顶点的数据直接在顶点渲染器中计算出最终全局坐标。每个顶点单独计算，先根据顶点所在关节点的控制曲线以及变换矩阵，然后根据该顶点在关节点变形模型网格中的序号计算其实际位置，并变换到全局坐标系中。

比较这两种实现方法，二次渲染实现的加速计算，使用顶点渲染器和片段渲染器分别计算关节点的和顶点的数据，可以避免关节点数据的重复计算，但在计算空间上需要申请 FBO 对象和 FBO 绑定纹理空间，在计算时间上需要渲染目标设置和数据回传两步骤。而一次渲染方法每个关节点的网格含有多个网格顶点，会出现重复计算，增加了总计算量，但该方法只需要一次渲染循环，而且不需要回传数据，提高了变形渲染执行的连贯性。

上述两种方法的优劣，取决于每个关节点的自适应网格中的顶点数目。当顶点数较少时，第二种方法速度较快，当顶点数达到一定数目，重复计算所需时间超过数据回传时间，则第一种方法速度较快。在本章的计算中，每个关节点的网格一般包括 20 ~ 40 个顶点，如果多个角色同时进行变形计算时，顶点个数更多，因而第二种方法计算速度较快。

## 3.3.2　相邻自适应网格的无缝连接

基于自适应网格的图像变形方法中，各个关节点的变形计算是互相独立的，通过网格的自动形状调整以实现角色的局部区域变形。在变形前，变形角色会根据关节点划分为一个个变形区域，这些区域的边界是邻接的，变形前的独立区域肯定是良好连接的，而变形后的相邻关节点的自适应网格能否实现良好的连接，是验证变形方法是否成功的不可或缺的步骤。

图 3-6 所示为相邻自适应网格无缝连接示意图，图中 $S_1S_2S_3S_4$ 和 $S_5S_6S_7S_8$ 分别是两个独立的自适应网格覆盖区域，分别在以关节点 $O_1$ 和 $O_2$ 为原点的局部坐标系中，以双曲线 $P_1P_2$ 和 $P_3P_4$ 为自适应网格中心双曲线，两网格的计算过程互不影响。对于变形区域 $S_1S_2S_3S_4$，$O_1$ 为该区域的当前关节点，$O_1P_1$ 和 $O_1P_2$ 分别

为两侧的骨骼，$S_1S_2$ 和 $S_3S_4$ 分别为其最外侧的分割映射线，其定义决定它们分别与两侧的骨骼线垂直。$S_5S_6S_7S_8$ 是该关节点相邻的下一个关节点的变形区域，这两个区域共用同一条骨骼 $O_1O_2$，所以 $O_1P_2$ 和 $O_2P_3$ 属于同一条直线，而 $P_2P_3$ 是同一个点；$S_3S_4$ 和 $S_5S_6$ 均过 $P_2$ 点而垂直于直线 $O_1O_2$，因此 $S_3S_4$ 和 $S_5S_6$ 也是同一条直线，由此可证，两区域的边界线是重合的，两个区域是无缝连接的。

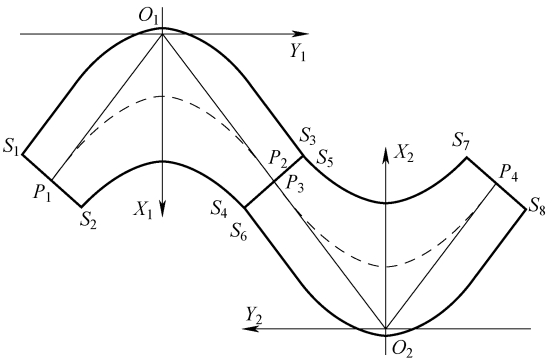



图 3-6　相邻自适应网格的无缝连接示意图

由上述理论可得，在变形过程中各个关节点的变形模型可以实现良好的衔接，能够完成整个角色的变形。

初步计算得到的自适应网格的顶点是在各关节点的局部坐标系中进行的，所有顶点坐标均为局部坐标，需要将各个顶点的局部坐标转换为全局坐标。局部坐标到全局坐标的转换矩阵，可根据各关节点的全局位置及其全局旋转角计算获得。一个关节点的全局旋转角，为与其在同一骨骼链并在其之前的所有关节点的旋转角之和。对于一条骨骼链，从其开始点开始，依次累加关节点旋转角，直至当前关节点，即可获得全局旋转角，同时根据其全局位置坐标，获得坐标转换矩阵，其下一关节点的全局位置，可由该全局旋转角和骨骼长度计算得出。根据坐标转换矩阵，自适应网格的顶点即可转换为全局坐标。

## 3.4　基于自适应网格的二维角色变形算法实现

自适应网格具有良好的可弯曲特性，可以自动根据当前的角色姿态改变网格形状，可以保持网格覆盖区域的总面积不变，可以实现各关节点间的网格平滑接合，而且可以实现硬件加速，因此十分适用于二维角色图像的变形。基于自适应网格的二维角色图像变形在变形结果真实感方面，可以通过控制曲线导数连续性，保证变形弯曲区域的平滑性；在变形计算实时性方面，通过影响因素局部化以及 GPGPU 图形硬件加速，保证实时变形计算。

基于自适应网格的二维角色变形过程可以分为如下几个步骤：

（1）获得变形角色的简化骨骼信息，这部分主要由用户手动交互完成；

（2）根据简化骨骼信息包括关节点位置、骨骼长度宽度和关节点旋转角；

（3）更新各关节点的自适应网格状态，并覆盖变形角色用于变形；

（4）交互控制变形，由用户控制角色关节点，当某关节点位置变化，其旋转角发生变化从而改变自适应网格形状，实现二维角色变形。

根据本章所述的变形方法思想，基于自适应网格的二维角色变形算法总流程如图 3-7 所示。

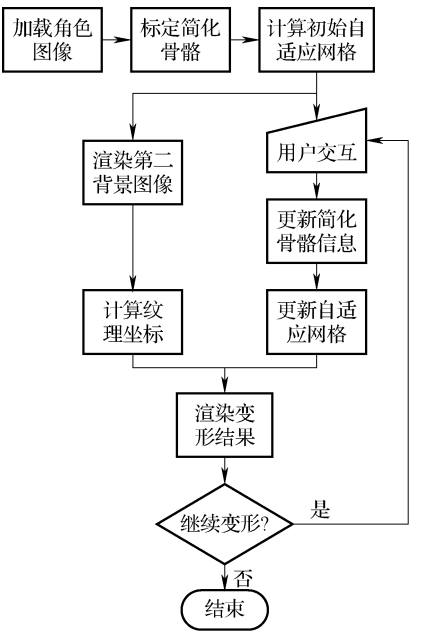



图 3-7　基于自适应网格的角色变形流程图

### 3.4.1　简化骨骼标定

变形过程的第一步，是获取角色变形区域的简化骨骼信息。骨骼信息可以使用多种方法获得，包括关节点位置及其连接关系，J. M. Lien[63] 详细描述了根据角色图像合理建立骨骼的方法。与其方法不同的是，自适应网格的构建只需要变形区域的简化骨骼框架即可，而不需要完整的骨骼框架，这大大简化了骨骼模型的建立过程。用户只要明确变形区域的位置，就可以方便地标定该角色的简化骨骼模型。

变形角色简化骨骼的标定由 3 步完成：

（1）按照角色的结构，标定需要变形部分的关节点位置，这些位置主要是肘

膝等旋转弯曲的位置点；

（2）连接每两关节点成为一条骨骼，互相连接的骨骼在本章中称为骨骼链，同一骨骼链上的关节点会互相影响，一个关节点位置移动，其他关节点会跟随移动；

（3）设置各段骨骼的宽度，变形区域的简化骨骼框架标定完成。标定过程中，用户可以随时调整关节点的位置或取消关节点。

### 3.4.2 变形过程初始化

变形过程的初始化可以由如下几步完成：

（1）构建角色简化骨骼框架。根据标定好的关节点位置及其连接关系，将独立的骨骼计算出首尾相连的关节点骨骼链，构建完毕；

（2）剔除变形区域的背景图像。根据初始状态的基本骨骼框架，计算对应初始状态的变形模型网格。以原图像角色为背景，以透明色渲染初始状态变形网格，创建第二背景图像，该图像已经去掉变形角色的变形区域，只保留非变形区域部分；

（3）获取变形模型纹理图。通过初始状态的变形模型网格，覆盖到初始角色图像上，提取 RGB 颜色信息，并根据像素颜色计算其透明因子 *alpha*，透明因子的计算方法为 $alpha = 0x\mathrm{FF} - R\&G\&B$。

### 3.4.3 交互变形及渲染

在交互变形过程中，需要完成如下几步：

（1）改变角色骨骼姿态。通过交互方式，或通过预定义动作驱动角色骨骼，计算各关节点的新位置，获得新的基本骨骼信息；

（2）重新计算变形模型网格。根据关节点的新位置，计算各关节点的旋转角，重新计算自适应网格，并连接各关节点的网格，获得对应当前状态的整体网格；

（3）渲染变形结果。以第二背景图像作为背景，以原始角色图像作为纹理，渲染当前变形模型网格，获得整个角色的变形结果。

## 3.5 二维角色变形关键代码

依据本书的自适应网格变形技术，已实现了二维角色变形的原型系统，该原型系统能够对输入的二维图像角色实现变形结果输出。在该原型系统中，以自适应网格的变形计算为中心，设立角色骨骼标记模块、自适应网格建立模块、自适应网格变形模块、纹理处理和变形结果输出模块，以及图形硬件加速接口模块。

整个系统中包含 21 个类，其中较为关键的模块类有 GPU 功能接口类、GPU 功能 CPU 模拟类、自适应网格变形控制类、自适应网格渲染类、OpenGL 环境搭建类、计时功能类等。

### 3.5.1　角色关节点数据结构

在自适应网格的建立和变形过程中，角色骨骼是一个最为关键的数据结构，包括动作的驱动、变形控制等均通过角色骨骼完成。角色骨骼数据由角色关节点、关节点连接关系组成，其中关节点的类定义表示如下：

```
class JPoint                          //关节点数据结构
{
public:
    int x,y;                          //关节点在角色图形中的相对位置
    int p2,pt,p3;                     //关节点前后点的索引号
    float alpha1;                     //关节点两侧骨骼线的夹角
    float rotAngle1,rotAngle2;        //关节点两侧骨骼线的偏向角
    float width1;                     //当前关节点所处位置的宽度
    float length2,length3;            //当前关节点前后骨骼的长度
    float lengthC;                    //当前关节点所控制的变形区域长度
    bool isLess180;                   //当前关节点的旋转角是否大于 180°
    bool isStart;                     //当前关节点是否为开始关节点
public:
    JPoint();
    bool operator == ();              //判定是否为同一关节点的运算符
    bool isCatched(CPoint point); //判定鼠标点中当前关节点
};
```

关节点连接关系的定义比较简单，在本原型系统中，仅通过一个有序向量对表示关节点的前后连接关系即可。

### 3.5.2　角色动作驱动部分代码

本书的二维角色变形算法，首先建立变形骨骼，然后根据当前的动作信息，使用图形硬件加速计算其变形网格，最后将获得的变形网格，通过三角带的方式渲染成最终的结果。

在动作驱动过程中，首先根据某控制关节点的新位置，更新当前关节点的变形信息，然后根据该关节点信息重计算其所在关节链的所有关节点的信息。根据动作信息更新当前关节点的信息的代码如下：

```
bool reCalPtData(int inx1)
{
    //改变的是 alpha1,rotAngle1,rotAngle2,isLess180
    SkeJPoint * pp1 = jointpoint + inx1;
    SkeJPoint * pp2 = jointpoint + pp1 -> p2;
    SkeJPoint * pp3 = jointpoint + pp1 -> p3;
    int p2vx,p2vy,p3vx,p3vy;
    p2vx = pp2 -> x - pp1 -> x;
    p2vy = pp2 -> y - pp1 -> y;
    p3vx = pp3 -> x - pp1 -> x;
    p3vy = pp3 -> y - pp1 -> y;
    pp1 -> isLess180 = (p2vy* p3vx - p3vy* p2vx) > = 0;
    //计算当前关节点的两侧骨骼夹角
    pp1 -> alpha1 = acosf((float)(p2vx* p3vx + p2vy* p3vy)/
        sqrtf((float)(p2vx* p2vx + p2vy* p2vy)* (p3vx* p3vx + p3
        vy* p3vy)))* 180.0f/3.14159265f;
    //借用超过 180°的条件，计算其前后骨骼的旋转角
    pp1 -> rotAngle1 = (pp1 -> isLess180? (180 - pp1 -> alpha1/2):
         - (180 - pp1 -> alpha1/2));
    pp1 -> rotAngle2 = (pp1 -> isLess180? - pp1 -> alpha1/2:
        pp1 -> alpha1/2);
    return true;
}
```

重新计算当前关节点之后同关节链所有关节点的信息代码如下：

```
bool reCalPtCoord(int inx1)
{
    //先找到该关节链
    int jNum = -1,cNum = -1;
    findPtInChains(inx1,cNum,jNum);
    //inx1 关节点，是第 cNum 条关节链中的第 jNum 个关节点
    //已找到关节链，修改该关节链中的关节点坐标
    float  angleSum = 0;
    float dx,dy;
    int * ptList = chain1[cNum].ptIndex;
    SkeJPoint * curjpt,* nextjpt;
```

```
    for(int i = 0; i < jNum; i + + )
    {
        curjpt = jointpoint + ptList[ i ];
        angleSum + = curjpt -> rotAngle1 + curjpt -> rotAngle2;
    }
    const float ratio1 = 3.14159265f/180.0f;
    float lenBone,angle1;
    for(int i = jNum; i < chain1[ cNum ].ptNum - 1; i + + )
    {
        curjpt = jointpoint + ptList[ i ];
        nextjpt = curjpt + 1;
        angleSum + = curjpt -> rotAngle1 + curjpt -> rotAngle2;
        lenBone = curjpt -> length3 + curjpt -> lengthC +
              nextjpt -> length2 + nextjpt -> lengthC;
        angle1 = angleSum* ratio1;
        dx = lenBone* cosf(angle1);
        dy = lenBone* sinf(angle1);
        nextjpt -> x = curjpt -> x + dx;
        nextjpt -> y = curjpt -> y + dy;
    }
    return true;
}
```

### 3.5.3　图形硬件 GPU 加速接口程序代码

在本章中支持二维角色变形的自适应网格，由于具有局部运算特性，因此可使用硬件加速特性以获得更佳的运行效率，为下一步大规模群组运动角色仿真提供支持。

在基于 GPGPU 的硬件加速算法中，图形硬件是通过图形 API 来调用其功能的。在进行加速运算之前，首先通过图形 API 初始化图形硬件，然后将 CG 程序分别加载到图形硬件的顶点渲染器和片段渲染器中。在计算过程中，首先将计算所需要的数据，由内存加载至显存中，并在合适的时机，设定某段渲染器程序有效，即可启用图形硬件加速功能，最终获得加速计算的结果，最终结果将通过图形 API 返回内存或输出至屏幕帧缓冲区。

图形硬件 GPU 的接口程序包括初始化运行环境、编译与编译 GPU 渲染器程序、调用 GPU 计算和结果返回几个部分。

**1. 初始化 GPU 运行环境**

初始化 GPU 运行环境的步骤是首先创建运行环境，然后获取 GPU 所能支持的最佳顶点程序配置和片段程序配置，最后设置顶点程序配置和片段程序配置，具体代码如下：

```
bool cgInitEnv()
{
    //为 Cg 程序创建新的运行环境
    cgContext = cgCreateContext();
    //判定运行环境是否创建成功
    if(cgContext == NULL)
    {
        TRACE("Failed To Create Cg Context");
        return false;
    }
    //获取当前显卡 GPU 所能支持的顶点程序配置
    cgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
    //确认配置是否获取成功
    if(cgVertexProfile == CG_PROFILE_UNKNOWN)
    {
        TRACE("Invalid Vertex profile type");
        return false;
    }
    //设置刚才获取的显卡最高顶点配置
    cgGLSetOptimalOptions(cgVertexProfile);
    //获取当前显卡所支持的最高片段程序配置
    cgFragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    //判断顶点配置是否获取成功
    if(cgFragmentProfile == CG_PROFILE_UNKNOWN)
    {
        TRACE("Invalid Fragment profile type");
        return false;
    }
    //设置刚才获取的最高片段配置
    cgGLSetOptimalOptions(cgFragmentProfile);
    return true;
}
```

**2. GPGPU 通用计算环境初始化**

GPGPU 通用计算环境初始化包括建立 GPGPU 计算环境和配置 FBO 离屏渲染对象两部分。建立 GPGPU 计算环境代码如下：

```
boolinitGlew()
{
    //建立 GPGPU 计算环境
    int err = glewInit();
    //判断计算环境是否建立成功
    if(GLEW_OK! =err)
    {
        TRACE((char* )glewGetErrorString(err));
        return false;
    }
    return true;
}
```

配置 FBO 离屏渲染对象代码如下：

```
bool initFBO(GLuint &fb,int texWidth,int texHeight)
{
    //生成并绑定一个 FBO，也就是生成一个离屏渲染对象
    if(fb == 0)    glGenFramebuffersEXT(1,&fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
    //视口的比例是 1：1，像素、纹理、数据三者一一对应
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0,texWidth,0,texHeight,10, -10);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glViewport(0,0,texWidth,texHeight);
    return true;
}
```

**3. 加载 GPGPU 计算程序**

在图形硬件中，有两个渲染器可以进行编程功能调用，它们分别是顶点渲染器和片段渲染器，在这两个渲染器中，分别运行顶点程序和片段程序，这两个程序需要在 GPGPU 计算之前提前进行编译并加载到渲染器中。

GPU 顶点渲染器程序编译和加载代码如下：

```
bool cgLoadVProgram(char * fileName,char * FuncName,char* * args)
{
    //加载并编译顶点渲染器程序
    cgVertexProgram = cgCreateProgramFromFile(cgContext, CG_SOURCE,
            fileName,cgVertexProfile,FuncName,NULL);
    //判定是否编译成功
    if(cgVertexProgram == NULL)
    {
        //编译出错，判定出错位置
        CGerror Error = cgGetError();
        //根据出错代码，输出出错信息
        TRACE(cgGetErrorString(Error));
        return false;
    }
    ASSERT(cgVertexProgram! = NULL);
    //将该顶点渲染器程序加载至显卡渲染器
    cgGLLoadProgram(cgVertexProgram);
    return true;
}
```

GPU 片段渲染器程序编译和加载代码如下：

```
bool cgLoadFProgram(char * fileName,char * FuncName,char* * args)
{
    //加载并编译片段渲染器程序
    cgFragmentProgram = cgCreateProgramFromFile(cgContext,
        CG_SOURCE,fileName,cgFragmentProfile,FuncName,NULL);
    //判定是否编译成功
    if(cgFragmentProgram == NULL)
    {
        //获取编译出错代码
        CGerror Error = cgGetError();
        //根据出错代码，输出出错信息
        TRACE(cgGetErrorString(Error));
        return false;
    }
    ASSERT(cgFragmentProgram! = NULL);
```

```
    //将该片段渲染器程序加载至显卡渲染器
    cgGLLoadProgram(cgFragmentProgram);
    return true;
}
```

**4. GPGPU 功能调用与结果取回**

GPGPU 计算的程序的编译和加载一般在初始化阶段进行，在场景渲染过程中，才进行功能调用，计算完毕之后，将计算结果返回至内存，以进行进一步处理。在功能调用过程中有如下步骤：设置绘画模式与目标、设置程序有效、设置程序失效和结果返回。设置绘画模式与目标的代码如下：

```
bool gcFrameInit()
{
    initFBO(fbo,texDivNum+1,texJptNum* 100);
    CGparameter modelViewMatrix;
    modelViewMatrix = cgGetParam(true,0,"ModelViewProj");
    cgGLSetStateMatrixParameter(modelViewMatrix,
        CG_GL_MODELVIEW_PROJECTION_MATRIX,
        CG_GL_MATRIX_IDENTITY);
    //直接描画绘制模式
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
    //把当前的 FBO 对象，与 FBO 纹理绑定在一起
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fbo);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
           GL_COLOR_ATTACHMENT0_EXT,
           GL_TEXTURE_RECTANGLE_ARB,TresultID,0);
    glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
    return true;
}
```

设置渲染器程序有效及取消代码如下：

```
bool gcFrameRender()
{
    //使用之前获取的顶点程序配置，并启用当前顶点渲染器程序
    cgGLEnableProfile(cgVertexProfile);
    cgGLBindProgram(cgVertexProgram);
```

```
    //使用之前获取的片段程序配置，并启用当前顶点渲染器程序
    cgGLEnableProfile(cgFragmentProfile);
    cgGLBindProgram(cgFragmentProgram);

    //    ……
    //普通模型渲染算法
    //    ……

    //计算完毕后，关闭顶点程序配置和片段程序配置
    cgGLDisableProfile(cgVertexProfile);
    cgGLDisableProfile(cgFragmentProfile);
}
```

GPGPU 计算完毕后，需要将计算结果返回至内存，以进行下一步处理：

```
bool gcReturnData()
{
    //从帧缓冲中读取数据，并把数据保存到 Data 数组中。
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0,0,texDivNum,texJptNum,GL_RGBA,GL_FLOAT,Data);
    //还原帧缓存和双缓冲描画模式
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,0);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    return true;
}
```

## 3.6 实验结果与分析

基于本章提出的二维角色变形思想与方法，现已经在 PC 上实现了一个基于自适应网格的二维角色变形演示程序。本章实验的硬件配置为 Intel P4 3.0G CPU，1GB 内存，nVidia GeForce FX6600 128M 显卡，软件环境为 Windows XP 操作系统，编程环境为 Microsoft Visual Studio 2005，3D 开发环境为 OpenGL。

本节实验使用的角色图像分别是① Igarashi[15] 论文中的 Frog 图像；② Frog2 图像；③ Girl 图像；④ Sketch。这些角色的简化骨骼信息见表 3-2。

表 3-2　角色图像简化骨骼信息表

| 角色图像 | 图像大小 | 关节点数 | 骨　骼　数 |
|---|---|---|---|
| Frog | 1024 × 1024 | 12 | 8 |
| Frog2 | 512 × 512 | 14 | 10 |
| Girl | 1024 × 1024 | 17 | 13 |
| Sketch | 512 × 512 | 14 | 10 |

本节的实验主要包含如下几部分：

（1）验证自适应网格对变形结果平滑性的贡献。本章提出的基于自适应网格的二维角色变形方法中，通过自适应网格将变形失真均匀地分布到整个变形区域中，从而降低变形失真。通过比较本章方法和其他方法的变形结果，验证本章方法在降低变形失真方面的有效性。

（2）获得自适应网格精细度对变形结果的影响。自适应网格是本章的变形方法中的主要辅助结构，其网格精细度直接影响变形失真的分布均匀性。通过比较不同网格精细度下的变形结果，说明网格精细度与变形结果的关系。

（3）获得变形计算时间与网格精细度的关系。本章的变形方法，使用图形学的方法实现图像的变形，网格精细度对应的是自适应网格中的顶点数量，顶点数量的增多，将增加求解的计算量。通过分别统计网格精细度不同情况下的计算时间，获得网格精细度对运行时间的影响，并同时验证图形硬件加速的有效性。

（4）获得变形计算时间与变形角色个数的关系。本章提出的基于自适应网格的二维角色变形算法，目的是为了适用于多角色的同时变形，本方法所能实时变形的角色数量是本文方法的重要衡量标准。通过分别统计在不同变形角色个数下的运行时间，说明角色个数对运行实时性的影响，并验证图形硬件加速的有效性。

### 3.6.1　基于自适应网格的二维角色变形结果

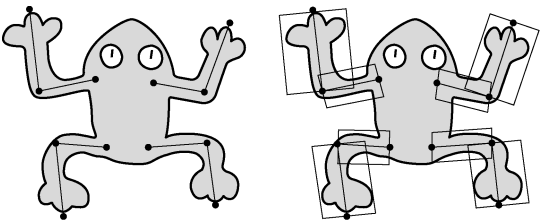
基于自适应网格的二维角色变形方法，通过自适应网格的平滑变形特性，实现角色图像的平滑变形，在本实验中主要检验自适应网格的平滑变形特性，通过与 Capel[57] 的图像变形方法以及 Igrarashi[15] 的方法比较，验证该方法用于二维角色变形的有效性以及对提高变形结果真实性的贡献。

**实验 1：Frog 实验结果及分析**

本节的第一组实验是使用 Igrarashi[15] 文章中的 Frog 图像作为原始角色，说明简化骨骼信息的获取、自适应网格构建方法，并通过与其他变形方法进行比较，说明本方法可以避免一些不合理结果的特性。

对于 Frog 角色图像来说，其主要变形区域主要在其四肢部分，针对这几部分划定简化骨骼。该角色的其简化骨骼信息如图 3-8 所示，其中图 3-8a 所示为 Frog

角色中的关节点及其连接关系示意图，图 3-8b 所示为该角色的各段骨骼宽度标定示意图。

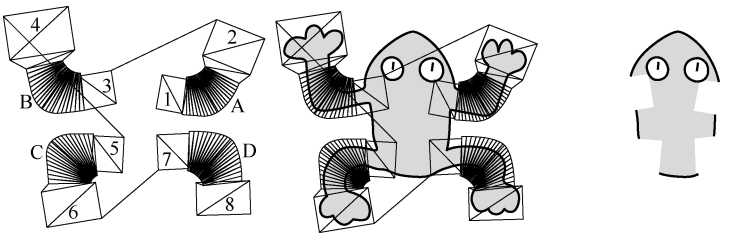


a)　　b)

图 3-8　Frog 简化骨骼示意图

a）关节点及其连接关系　b）简化骨骼框架

Frog 角色的简化骨骼包含 12 个关节点和 4 条骨骼链，该角色的自适应网格状态如图 3-9 所示，其中图 3-9a 所示为初始状态的自适应网格，图中三角带区域 *ABCD* 为完整的四个非首尾关节点的自适应网格，区域 1 ~ 8 是 8 个首尾关节点退化了的自适应网格，只保留了一个长度补足矩形。图 3-9b 所示为初始状态的自适应网格覆盖在角色图像上的效果，可以看出所有关节点的自适应网格将变形区域包含其中。图 3-9c 是使用自适应网格进行剔除变形区域图像的第二背景图像。




a)　　b)　　c)

图 3-9　Frog 角色的自适应网格

a）自适应网格　b）自适应网格的覆盖效果　c）第二背景图像

本章的变形方法应用于 Frog 图像的变形结果如图 3-10 所示，对于该图像，变形区域为肘与膝盖区域，而该区域的角色图像宽度小、单位长度面积贡献小，容易发生尖角等失真。图 3-10 a 所示为原图像，图 3-10b 所示为 Steve Capel 方法的变形结果，从图中可以看出，对于肘关节的弯曲，Steve Capel 的方法出现很明显的尖角问题，这是由于没有综合考虑各区域之间的过渡关系造成的。图 3-10c 所示为 Igarashi 的变形结果，该方法在肘部弯曲时也出现不真实的结果，这是由于肘部关节处顶点较少，占的权值较小，最小化求解自由顶点的位置时过度变形造成的。图 3-10d 所示为本章变形方法的变形结果，从图中可以看出，对于同一

个变形动作来说，本章的方法能够较好地实现骨骼角色变形，在肘关节和膝盖等部位避免尖角等失真。

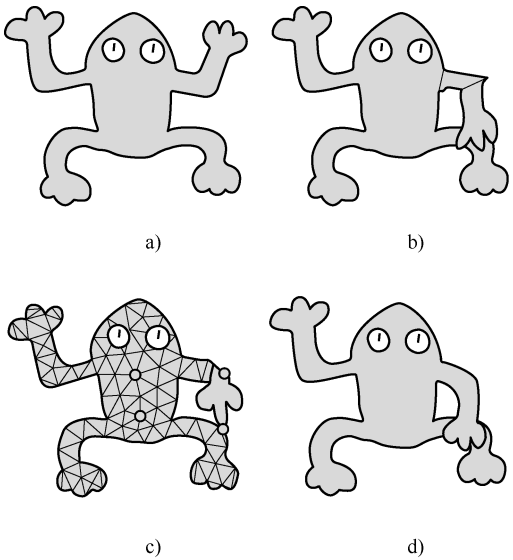



图 3-10　本章与其他方法的 Frog 变形结果比较 1

a）原图像　b）Capel 方法的结果

c）Igarashi 方法的结果　d）本章方法的结果

如图 3-11 所示为 Igarashi 方法与本章所提出的方法应用到 Frog 角色上的另一组变形结果比较，由图中可以看出，本方法可以比 Igarashi 方法能够产生更平滑的变形结果。

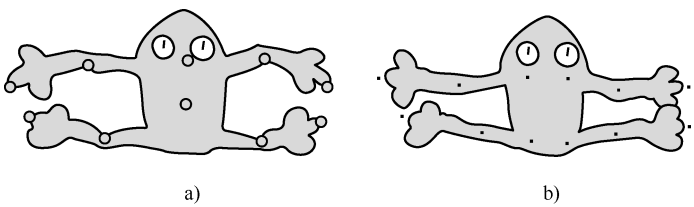



图 3-11　本章与其他方法的 Frog 变形结果比较 2

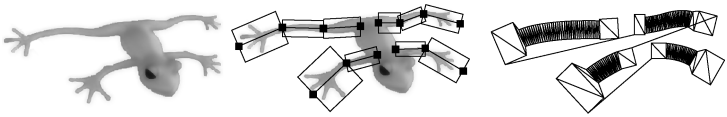
a）Igarashi 方法的结果　b）本章方法的结果

**实验 2：Frog2 实验结果及分析**

本章的第二组实验是使用从网络中获取的图片 Frog，为与上一个实验中的 Frog 区分，重新命名为 Frog2。通过使用 Capel 的 FFD 变形方法、Igarashi 的变形方法以及本章提出的基于自适应网格的变形方法，对该角色图像进行变形，比较

其变形结果，说明本方法在降低变形结果失真方面的有效性。

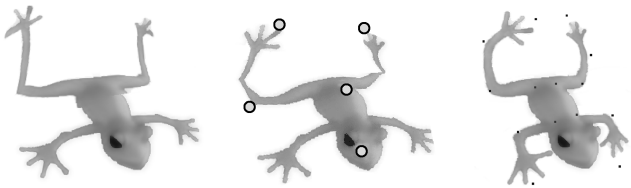
Frog2 角色的简化骨骼与自适应网格如图 3-12 所示，使用该变形网格，实现的变形结果如图 3-13c 所示，图 3-13a 和图 3-13b 分别是 Capel 方法的变形结果和 Igarashi 方法的变形结果，从图中可以看出本章的方法可以提供更平滑的、失真度更小的变形结果，可以有效地避免关节点周围的尖角等不合理结果。



a)　　b)　　c)

图 3-12　Frog2 的简化骨骼和自适应网格

a）原图像　b）简化骨骼　c）自适应网格



a)　　b)　　c)

图 3-13　不同方法的 Frog2 变形结果比较

a）Capel 方法的结果　b）Igarashi 方法的结果　c）本章方法的结果

**实验 3：Girl 实验结果及分析**

本章的第三组实验是使用从网络中获取的实验图片 Girl，不同方法对 Girl 图像的变形结果如图 3-14 所示，图 3-14a 所示为原图像，图 3-14b 所示为 Capel 方法的变形结果，从图中可以看出，对于膝盖关节这样的弯曲仿真，该方法将出现尖角问题。图 3-14c 为 Igarashi 的变形结果，该方法在膝盖处出现变细的现象，这是由于他的方法使用的是全局面积保持，而不是局部面积保持。图 3-14d 所示为本章变形方法应用于 Girl 角色的变形结果，从图中可以看出，对于同一个变形动作来说，本章的方法能够较好地实现骨骼角色变形，在肘关节和膝盖等部位避免尖角等失真。

**实验 4：其他角色的实验结果**

本章的第四组实验使用从网络中获取的另外三幅图片，应用基于自适应网格的变形方法进行变形，进一步说明本章方法的有效性和适应性，变形结果如图 3-15 所示。图 3-15a、图 3-15d 和图 3-15f 分别为来源于网络的 Sketch 图片、Deer 图片和 Dancer 图片，从这些图片的变形结果中可以进一步说明本章变形方法的适用性。

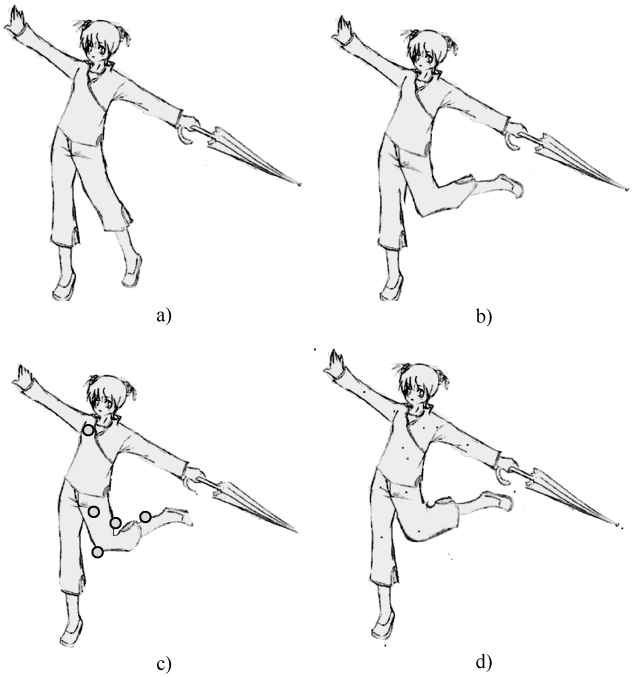



图 3-14　不同方法的 Girl 变形结果比较

a）原图像　b）Capel 方法的结果　c）Igarashi 方法的结果　d）本章方法的结果

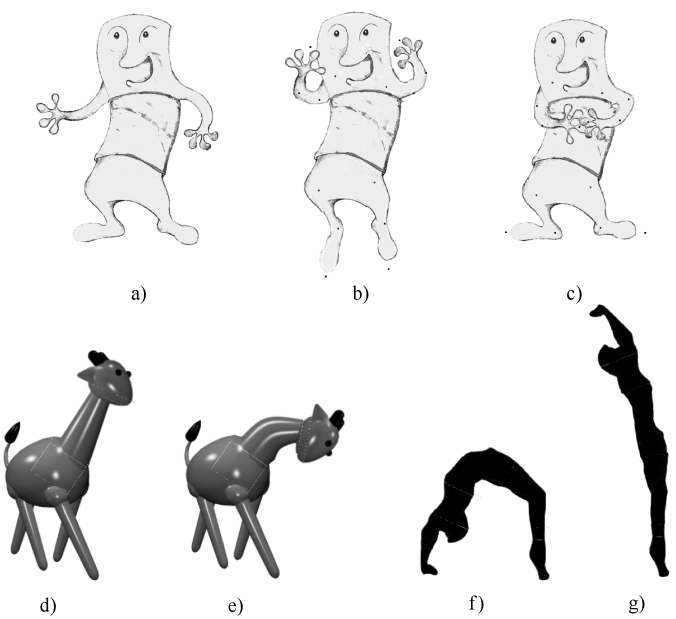



图 3-15　其他角色变形结果

a）Sketch 原图像　b）本章方法的变形结果　c）本章方法的变形结果

d）Deer 原图像　e）本章方法的变形结果　f）Dancer 原图像　g）本章方法的变形结果

### 3.6.2　自适应网格精细度对变形结果的影响

在基于自适应网格的二维角色变形方法中，自适应网格的精细程度对变形结果具有较大的影响，理论上自适应网格的精细程度越高，则变形的失真将越分散，整体失真度越低。本节使用 Frog 图像作为实验图片，通过比较不同精细程度的变形结果，说明自适应网格精细度对变形结果的影响。

**实验 1：Frog 实验结果及分析**

不同自适应网格精细度的 Frog 变形结果比较如图 3-16 所示，其中图 3-16a 为三角形分割数为 4 的自适应网格初始状态，图 3-16b 和图 3-16c 是在该精细程度的自适应网格支持下的变形结果，圈点着重标识处为该角色的主要失真部位。图 3-16d 和图 3-16g 分别是自适应网格的三角形分割数为 12 和 20 时的网格的初始状态，图 3-16e、f 和图 3-16g、h 分别是自适应网格的三角形分割数为 12 和 20 时的变形结果。

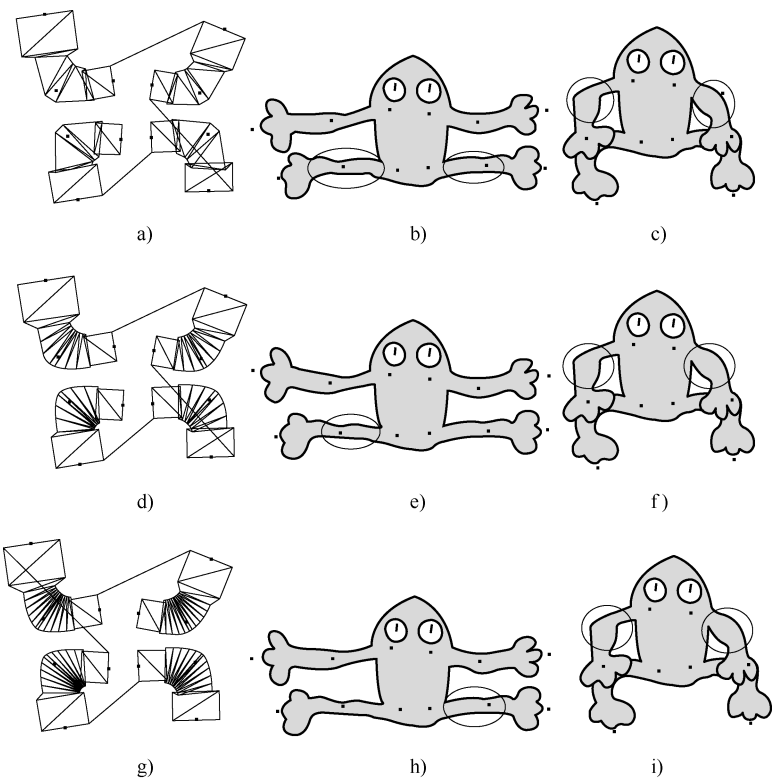



图 3-16　不同网格精细度的 Frog 变形结果比较 1

a）三角形分割数为 4 的自适应网格　b）变形结果 1　c）变形结果 2
d）三角形分割数为 12 的自适应网格　e）变形结果 3　f）变形结果 4
g）三角形分割数为 20 的自适应网格　h）变形结果 5　i）变形结果 6

从变形结果的圈点着重标识处可以看出，在其他条件相同的前提下，随着三角形分割数的增多，关节点周围的失真越来越小，这是由于变形失真被均匀分布到整个变形区域的结果，角色图像的变形变得平滑。

图 3-17 分别所示为自适应网格的三角形分割数为 40 和 60 的自适应网格外观及变形结果截图，从图中可以看出越精细的自适应网格，变形结果的失真也越小。但三角形分割数目为 40 是一个分界点，更为精细的自适应网格所能带来的增益很小。精细程度越高，带来的计算量也越大，因此在针对该角色图像的变形计算中，可以选择 40 个三角形的自适应网格。当对变形速度要求较高时，可以适当再降低自适应网格的精细程度。

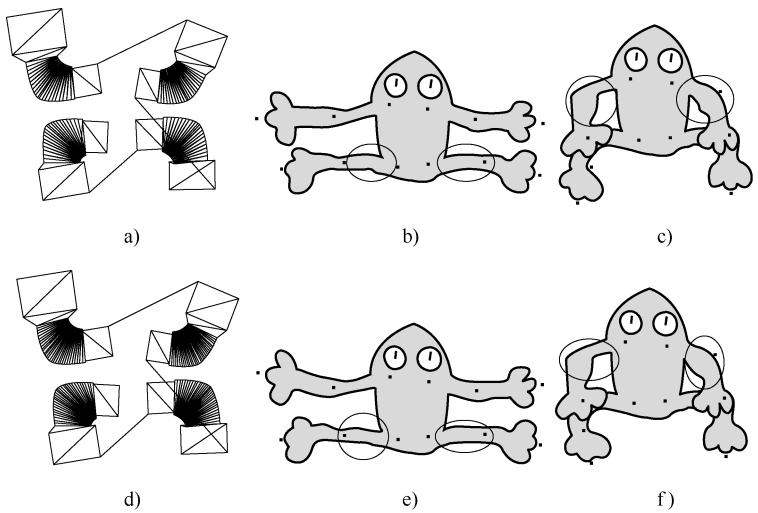



图 3-17　不同网格精细度的 Frog 变形结果比较 2

a）三角形分割数为 40 的自适应网格　b）变形结果 1　c）变形结果 2

d）三角形分割数为 60 的自适应网格　e）变形结果 3　f）变形结果 4

**实验 2：Frog2 实验结果及分析**

图 3-18 是针对 Frog2 图像，使用不同精细度的自适应网格的变形结果比较，

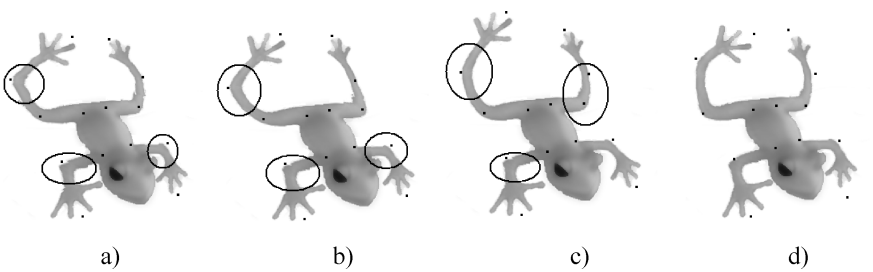



图 3-18　不同网格精细度的 Frog2 变形结果比较

a）4 三角形　b）12 三角形　c）20 三角形　d）40 三角形

图 3-18a、图 3-18b 和图 3-18c 是分别使用三角形分割数为 4、12 和 20 的自适应网格的变形结果，通过着重标识处可以看出，随着网格精细度提高，变形结果的平滑性也相应提高。更高精细度的变形结果，如图 3-18d 的三角形分割数为 40，即每个关节点对应的自适应网格中，包含 40 个三角形。

### 3.6.3 变形计算时间与网格精细度的关系

从上节给出的变形效果图中可以看出，在其他条件相同的前提下，越精细的自适应网格，变形结果的失真也越小，但精细程度越高的自适应网格计算，所需的计算量也越大。在运动角色仿真应用中，不仅需要变形结果的真实性，还需要变形速度的高效性。

在本节的实验中，以 Frog 角色的图像为例，获得各步骤运行时间与自适应网格的精细度的关系。为更准确地统计精细程度的不同造成的时间差，使角色数量为 100 个，这样可以减小单个角色变形计算时的统计误差。在时间统计过程中，每次实验连续记录 100 帧的运行时间，并求其平均值。

（1）在基于自适应网格的图像变形方法中，初始化步骤包含初始化运行环境、加载角色图像、准备简化骨骼数据、计算自适应网格的顶点纹理坐标等操作。在不同的网格精细程度，所需要的初始化各步骤所需时间见表 3-3，由表中可以看出，随着网格三角形划分数的增多，纹理坐标的计算数量开始渐渐增加，这是由于三角形数量增多，顶点的纹理坐标计算量增加的缘故。

**表 3-3 初始化步骤所需时间** （单位：ms）

| 自适应网格<br>三角形划分数 | 运行环境初始化<br>与数据准备 | 角色图像加载 | 纹理坐标计算 |
|---|---|---|---|
| 4 | 0.05190 | 180.6 | 0.2010 |
| 8 | 0.05195 | 177.9 | 0.2608 |
| 12 | 0.07875 | 182.8 | 0.2665 |
| 20 | 0.04795 | 183.5 | 0.3001 |
| 40 | 0.05698 | 179.6 | 0.3027 |
| 60 | 0.05749 | 193.2 | 0.4327 |
| 80 | 0.07261 | 180.3 | 0.4863 |
| 120 | 0.1212 | 183.7 | 0.5179 |

本节的实验对于本章提出的图像变形的硬件加速方法，分别实现了 CPU、GPU 二次渲染和 GPU 一次渲染三种加速算法，统计其运行时间并进行比较。对于网格精细度对变形过程中的各步骤运行时间影响，通过记录 100 个 Frog 角色变形过程中的数据，每次实验连续记录 100 帧，取其平均值。使用 CPU 计算 100 个

Frog 角色的变形，共做三组实验，各步骤运行时间见表 3-4。

**表 3-4　传统 CPU 方法运行时间**　　（单位：ms）

| 自适应网格三角形划分数 | 自适应网格顶点计算时间 1 | 自适应网格顶点计算时间 2 | 自适应网格顶点计算时间 3 |
|---|---|---|---|
| 4 | 20. 83 | 20. 69 | 21. 56 |
| 8 | 24. 45 | 24. 89 | 24. 62 |
| 12 | 30. 09 | 29. 35 | 30. 64 |
| 20 | 36. 77 | 37. 06 | 38. 61 |
| 40 | 53. 75 | 53. 85 | 53. 80 |
| 60 | 71. 95 | 72. 75 | 71. 60 |
| 80 | 89. 2 | 89. 15 | 101. 25 |
| 120 | 125. 0 | 123. 6 | 123. 7 |

根据表 3-4 中的数据所绘制的传统 CPU 方法变形计算时间与网格精细度的关系如图 3-19 所示。通过图中可以看出，网格顶点的计算时间随顶点增多而增加的趋势，CPU 是直接计算方式，两者基本成正比关系（图中横轴为非等分尺）。

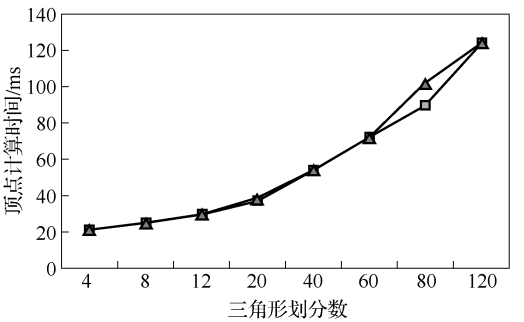



图 3-19　CPU 方法的运行时间与网格精细度关系图

（2）自适应网格可以使用 GPGPU 技术进行图形硬件加速，大大加快网格顶点的计算速度。当前主流的硬件配置下，单纯以浮点运算速度为指标，GPU 的计算速度要远远高于 CPU，但对于二次渲染方法来说，GPU 每次计算过程分为初始化、计算和数据取回三部分，初始化操作包括设置渲染目标以及上传输入数据等，重置操作包括重置渲染目标，以及将计算结果数据从 FBO 缓冲区取回到内存等。

使用与 CPU 方法同样的运行条件，通过 GPU 二次渲染方法进行网格顶点加速计算，连续记录运行数据，所获得的 GPU 二次渲染方法变形过程中各步骤运行时间见表 3-5。

表 3-5　GPU 二次渲染方法各步骤运行时间　（单位：ms）

| 自适应网格三角形划分数 | 帧加速计算环境初始化时间 | 网格顶点计算时间 | 帧加速计算数据返回时间 | 每帧计算总时间 |
|---|---|---|---|---|
| 4 | 0. 2311 | 0. 7121 | 0. 5837 | 1. 527 |
| 8 | 0. 2382 | 0. 7237 | 0. 8345 | 1. 796 |
| 12 | 0. 2310 | 0. 7111 | 0. 9345 | 1. 877 |
| 20 | 0. 2495 | 0. 7616 | 1. 304 | 2. 315 |
| 40 | 0. 2356 | 0. 8627 | 2. 354 | 3. 452 |
| 60 | 0. 2404 | 0. 9885 | 3. 117 | 4. 346 |
| 80 | 0. 2480 | 1. 124 | 4. 268 | 5. 640 |
| 120 | 0. 2534 | 1. 581 | 6. 224 | 8. 058 |

根据表 3-5 的数据绘制的 GPU 二次渲染法运行时间与网格精细度关系图如图 3-20 所示，通过图中可以看出，在 GPU 二次渲染方法中，环境初始化的时间一般会保持不变，而网格顶点计算时间会随自适应网络精细度增加而增加，数据取回时间也会随顶点数据的传输量增加而增加。

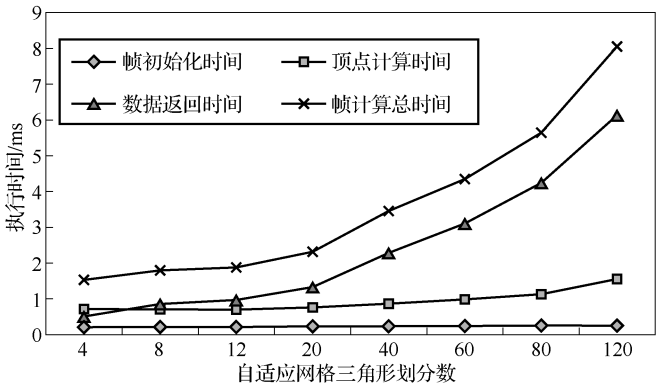



图 3-20　GPU 二次渲染法运行时间与网格精细度关系图

（3）在 GPGPU 加速方法中，二次渲染一般应用于复杂的计算过程，而一次渲染方法一般应用于较简单的几何属性计算。在图像变形过程中，GPGPU 加速计算的是自适应网格的顶点坐标，可以使用一次渲染方法进行加速。

使用 GPU 一次渲染方法统计各步骤运行时间，运行环境与上面 CPU、GPU 二次渲染条件相同，使用 100 个 Frog 角色图像同时进行变形计算，所获得的在不同网格精细程度下的运行时间数据见表 3-6。

表 3-6　GPU 一次渲染方法运行时间　　（单位：ms）

| 自适应网格<br>三角形划分数 | 网格顶点计算<br>时间 1 | 网格顶点计算<br>时间 2 | 网格顶点计算<br>时间 3 |
|---|---|---|---|
| 4 | 0. 7679 | 0. 7854 | 0. 7644 |
| 8 | 1. 053 | 1. 041 | 1. 029 |
| 12 | 1. 285 | 1. 322 | 1. 269 |
| 20 | 1. 819 | 1. 786 | 1. 837 |
| 40 | 3. 542 | 3. 501 | 3. 604 |
| 60 | 5. 347 | 5. 399 | 5. 453 |
| 80 | 6. 269 | 6. 223 | 6. 271 |
| 120 | 8. 285 | 8. 172 | 8. 166 |

根据表 3-6 中的数据所绘制的 GPU 一次渲染方法运行时间与网格精细度的关系如图 3-21 所示，通过图中可以看出，随着自适应网格精细程度的提高，变形计算量增大，从而造成 GPU 一次渲染法加速计算的时间变长。

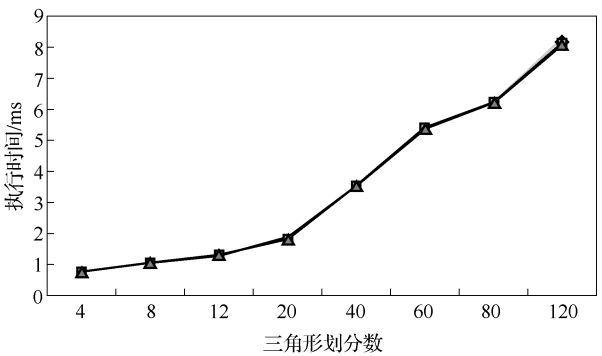



图 3-21　GPU 一次渲染方法运行时间与网格精细度关系图

（4）本节的变形算法包括传统 CPU 计算、GPU 二次渲染方法加速和 GPU 一次渲染方法加速三种，分别统计网格精细度对各种方法的运行时间影响。这三种执行方法所需的变形计算总时间随网格精细程度的变化如图 3-22 所示，从图中可以看出，GPU 相比 CPU 可以提供显著的加速作用，验证了针对二维角色变形的硬件加速算法的有效性。

不同网格精细度的 GPU 二次渲染与 GPU 一次渲染方法的对比如图 3-23 所示，图中可以看出两者的计算时间是相似的，造成这种现象的原因主要有两点：一方面，GPU 二次渲染所需要上传的数据少，回传的数据量多，而 GPU 一次渲

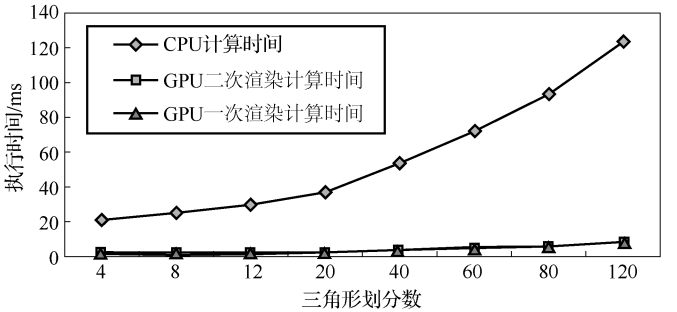



图 3-22　不同网格精细度的 CPU/GPU 变形计算时间比较

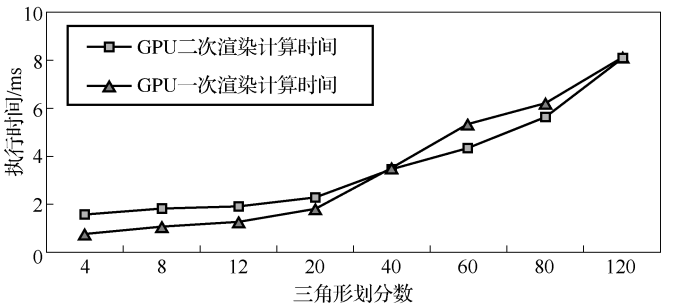



图 3-23　不同网格精细度的 GPU 方法的计算时间比较

染所需要上传的数据量多，GPU 二次渲染的回传数据量应该等于 GPU 一次渲染的上传数据量，都为自适应网格的顶点坐标数据量；另一方面，GPU 二次渲染避免重复计算，减少了计算量，而 GPU 一次渲染则避免了重复设置/还原渲染目标。

### 3.6.4　变形计算时间与角色个数的关系

本章提出的基于自适应网格的图像变形算法，是针对多角色的同时变形设计的，由于自适应网格的影响因素局部化和图形硬件加速，使得该算法的执行速度很快。

下面的实验以 Frog 角色图像为例，网格精细程度设定为每个自适应网格包含 40 个三角形，每次实验连续记录 100 帧取平均值，统计各步骤运行时间与角色个数的关系。

（1）统计角色个数不同所引起的初始化各步骤时间变化。各步骤运行时间见表 3-7。由表中可以看出，在角色变形过程中，由于对所有角色使用的是同一个图像，图像加载和纹理坐标计算的时间与角色数量无关，基本保持不变。

**表 3-7　变形初始化各步骤时间**　　（单位：ms）

| 参与变形的角色个数 | 运行环境初始化与数据准备 | 角色图像加载 | 纹理坐标计算 |
|---|---|---|---|
| 1 | 0. 07082 | 181. 7 | 0. 2171 |
| 5 | 0. 07073 | 177. 0 | 0. 2926 |
| 10 | 0. 07016 | 178. 4 | 0. 3658 |
| 20 | 0. 07112 | 181. 8 | 0. 3668 |
| 40 | 0. 07066 | 180. 1 | 0. 3731 |
| 60 | 0. 07085 | 183. 6 | 0. 3620 |
| 80 | 0. 07050 | 180. 6 | 0. 3681 |
| 100 | 0. 07085 | 179. 5 | 0. 3666 |
| 200 | 0. 07088 | 182. 9 | 0. 3663 |
| 300 | 0. 07096 | 180. 6 | 0. 3680 |
| 400 | 0. 07095 | 179. 6 | 0. 3717 |
| 500 | 0. 07059 | 180. 8 | 0. 3675 |

（2）统计角色个数不同引起的 CPU、GPU 二次渲染和 GPU 一次渲染等方法中运行时各步骤所需的时间变化。CPU 方法各步骤所需的运行时间统计见表 3-8。

**表 3-8　传统 CPU 方法所需的运行时间**　　（单位：ms）

| 参与变形的角色个数 | 自适应网格顶点计算总时间 1 | 自适应网格顶点计算总时间 2 | 自适应网格顶点计算总时间 3 |
|---|---|---|---|
| 1 | 0. 6055 | 0. 6085 | 0. 6035 |
| 5 | 2. 7855 | 2. 7785 | 2. 7335 |
| 10 | 5. 43 | 5. 42 | 5. 465 |
| 20 | 10. 81 | 10. 7 | 10. 795 |
| 40 | 21. 39 | 21. 51 | 21. 39 |
| 60 | 32. 065 | 32. 055 | 32. 62 |
| 80 | 42. 405 | 43. 255 | 43. 125 |
| 100 | 53. 35 | 53. 2 | 62. 4 |
| 200 | 107. 25 | 107. 05 | 107. 2 |
| 300 | 159. 55 | 159. 55 | 159. 35 |
| 400 | 212. 7 | 213. 4 | 215. 45 |
| 500 | 265. 2 | 266. 05 | 265. 75 |

根据表 3-8 所示的数据，获得的传统 CPU 方法变形计算的运行时间与角色个

数关系图如图 3-24 所示。

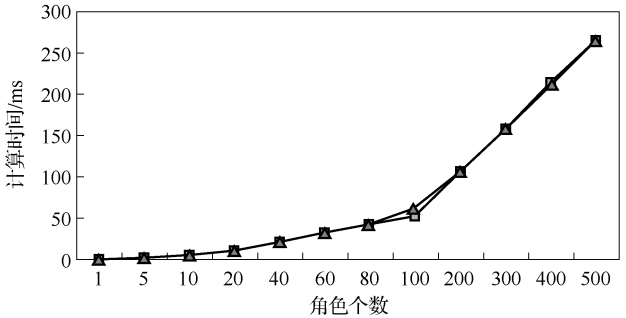



图 3-24　CPU 方法变形计算时间与角色个数关系图

（3）在 GPU 二次渲染方法的计算过程中，将角色网格顶点的计算结果暂时保存到显存的 FBO 纹理中，然后返回到内存进行二次渲染。FBO 纹理的每一行，对应角色的每一个关节点，本实验中的每个角色有 12 个关节点，由于本实验中使用的显卡，所支持的纹理最大为 4096 行，因此最大只能支持 4096/12 = 341 个 Frog 角色的同时变形。不同角色个数的 GPU 二次渲染方法中各步骤的运行时间统计见表 3-9。

**表 3-9　GPU 二次渲染方法中各步骤的运行时间**　（单位：ms）

| 参与变形的角色个数 | 帧加速计算环境初始化时间 | 网格顶点计算时间 | 帧加速计算数据返回时间 | 每帧计算总时间 |
|---|---|---|---|---|
| 1 | 0. 2207 | 0. 1394 | 0. 1996 | 0. 5597 |
| 5 | 0. 2177 | 0. 1637 | 0. 2866 | 0. 6681 |
| 10 | 0. 2206 | 0. 2004 | 0. 4031 | 0. 8241 |
| 20 | 0. 2185 | 0. 2664 | 0. 6343 | 1. 119 |
| 40 | 0. 2260 | 0. 4254 | 1. 073 | 1. 724 |
| 60 | 0. 2138 | 0. 5767 | 1. 511 | 2. 301 |
| 80 | 0. 2140 | 0. 7005 | 1. 919 | 2. 834 |
| 100 | 0. 2234 | 0. 8487 | 2. 333 | 3. 405 |
| 200 | 0. 2261 | 1. 591 | 4. 506 | 6. 322 |
| 300 | 0. 2379 | 2. 311 | 6. 600 | 9. 148 |

根据表 3-9 所获得的 GPU 二次渲染方法运行时间与角色个数关系如图 3-25 所示。在所支持的变形实验结果中可以看出，随着角色数量的增多，计算环境初始化时间基本保持不变，顶点计算时间逐渐增大，而数据返回时间与数据量基本成正比。

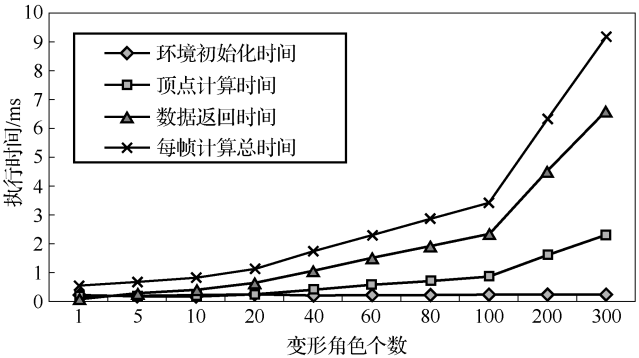



图 3-25　GPU 二次渲染方法运行时间与角色个数关系图

（4）GPU 一次渲染方法中各步骤的运行时间统计见表 3-10。

**表 3-10　GPU 一次渲染方法所需的运行时间**　　（单位：ms）

| 参与变形的角色个数 | 自适应网格顶点计算总时间 1 | 自适应网格顶点计算总时间 2 | 自适应网格顶点计算总时间 3 |
|---|---|---|---|
| 1 | 0. 1017 | 0. 09904 | 0. 1004 |
| 5 | 0. 2227 | 0. 2238 | 0. 2247 |
| 10 | 0. 3860 | 0. 3775 | 0. 3702 |
| 20 | 0. 6927 | 0. 7106 | 0. 7099 |
| 40 | 1. 284 | 1. 333 | 1. 338 |
| 60 | 1. 922 | 1. 962 | 1. 947 |
| 80 | 2. 484 | 2. 523 | 2. 513 |
| 100 | 3. 562 | 3. 638 | 3. 583 |
| 200 | 5. 195 | 5. 180 | 5. 165 |
| 300 | 8. 325 | 8. 480 | 8. 380 |
| 400 | 11. 56 | 11. 69 | 11. 63 |
| 500 | 14. 71 | 14. 875 | 14. 93 |

根据表 3-10 所得到的 GPU 一次渲染方法变形计算时间与变形角色个数的关系如图 3-26 所示。由图中可以看出，GPU 一次渲染方法的变形计算时间与变形角色个数基本成正比，这是由于该方法不需要每帧计算的初始化，与 CPU 一样只需要计算过程。

（5）根据上述实验中对 CPU、GPU 二次渲染和 GPU 一次渲染的变形计算的时间统计，可得三种计算方法随角色数量变化的执行时间对比如图 3-27 所示。

通过本节的实验可以看出，GPU 对于关节点数目较多的情况中有明显的计算

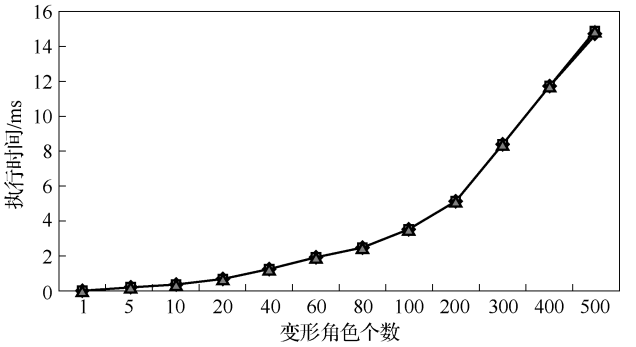



图 3-26　GPU 一次渲染方法运行时间与角色个数关系图

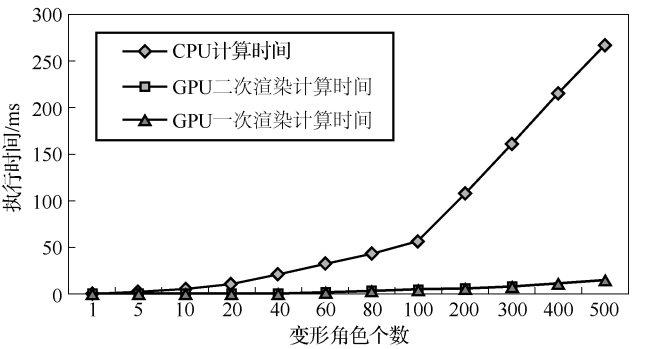



图 3-27　不同角色个数的三种方法计算时间比较

优势，而 GPU 一次渲染方法比 GPU 二次渲染方法的适用范围更广，GPU 二次渲染方法适合于大量顶点的计算，而 GPU 一次渲染加速方法适合于小计算量的硬件加速实现。对于 GPU 二次渲染加速方法，随着变形角色的个数少于 5 个时，CPU 的计算速度更快，当关节点个数超过 5 个时，GPU 的计算速度将很快超过 CPU，计算时间大大缩短，这是由于渲染目标的设置时间造成的，当渲染目标的切换时间大于 CPU/GPU 的计算差距，则 GPU 的每帧计算时间长，反之则 CPU 计算时间长。而对于 GPU 一次渲染加速方法，则计算速度始终快于 CPU。

## 3.7　本章小结

本章针对目前二维角色变形方法中变形结果容易出现不平滑失真和变形算法无法进行硬件加速的问题，提出了一种基于自适应网格的二维角色变形方法。该方法中采用的自适应网格操作简单，计算量小，可以实现二维角色快速变形。通过调节自适应网格的控制曲线参数，改变网格形状以适应角色姿态改变，利用网

格控制曲线的导数连续性避免变形中的尖角，实现二维角色的平滑变形效果；通过自适应网格的面积保持特性减小变形结果的形状失真；采用图形硬件加速自适应网格的顶点计算，进一步提高变形速度。

实验证明，本章提出的方法是可行的，并且对于解决图像变形中的不平滑失真与尖角问题是有效的。变形过程中所使用的 GPGPU 加速，有效地提高了模型变形速度。

进一步的工作，将研究动物角色变形过程中的合理性约束方法，并研究使用更少交互操作实现的角色变形方法。

# 第 4 章　三维运动角色变形技术研究

三维模型变形技术是计算机图形学中的重要技术之一，其广泛应用于角色动画、交互建模、受力仿真和医疗手术等领域中。随着扫描技术和数据可视化技术的发展，模型的获取途径越来越多，三维模型变形的应用也越来越广。计算机虚拟场景中的运动角色变形，根据变形对象的模型不同可以分为二维角色和三维角色两种类型。二维角色的绘制速度快，但由于图像信息量的缺失，对于某些动作的表达仍然存在困难，所能实现的变形动作数量有限，适用于角色动作较简单的情况；三维角色的模型表现力更为丰富，可以实现任意动作的变形，适用于角色动作较复杂的情况。

目前在计算机图形学领域中，研究者们已经提出许多针对三维几何模型的变形技术，如 FFD 技术[8]、基于骨骼变形技术[14]、Skining 变形技术[59]等，为提高变形真实度，细节保持的变形方法[16,76,79,81]成为主流的 3D 变形算法。2005 年 Zhou[121]中提出体积保持的大模型变形方法，开启了属性保持的变形方法先河。但对于大量角色的快速仿真应用，目前的变形方法仍然存在以下几方面的问题：

（1）变形算法复杂，变形速度慢，经过优化后的变形算法可实现单个模型的实时变形，但对于大模型变形或者多个目标同时进行变形则显得力不从心；

（2）变形方法对于细节、体积等模型属性的保持已经比较完善，但对于模型的表面积保持关注不足，而表面积保持是柔软物体和壳体变形的一大特征；

（3）算法不适合于硬件加速，这对于提高算法的计算速度是一个很大的限制。

本章针对三维物体模型的变形，提出了一种以规则网格作为基础模型的三维物体变形方法，由于该变形方法中针对不同的三维模型可以使用相同的基础模型，因此将规则网格的基础模型称为统一基础模型。该方法可以在变形过程中实现细节保持以及模型表面积保持，对柔软物体变形时可以仿真物体变形表面的褶皱现象；根据控制曲线将变形角色划分为独立的变形区域，对于每一局部变形区域，使用规则圆柱体网格作为变形基础模型，减少基础模型的生成计算步骤；使用阻尼振荡曲线实现物体表面的皱褶起伏，保持顶点列长度不变，实现模型表面积保持，同时仿真弯曲变形内侧的皱褶；通过以控制线为中心进行径向计算，实现细节模型的获取和合成，实现模型细节保持。实验证明，本方法能够有效仿真三维物体的变形，在细节保持的基础上，实现了表面积保持，并可实现柔软物体的表面皱褶等特征，提高变形结果的真实感，对于 1 万个顶点的三维模型，可以

达到 30FPS 的实时变形。

本章的其他部分将包含如下内容：

（1）介绍当前模型变形的各种变形方法，分析其中存在的问题；

（2）提出一种基于统一基础模型的变形方法，介绍其变形原理、表面积保持方法及细节保持方法；

（3）给出本章变形方法的算法实现；

（4）验证算法的可行性，并进行实验对比以及运行时间分析。

## 4.1　相关工作

在虚拟现实的表面模型中，几何形状的表达主要有参数曲面表示、多边形表示、子分平面表示和内隐表面表示等几种表示方式[34]，其中参数曲面和多边形常用于变形计算，多边形可以表示任意拓扑结构的物体，在模型建模中应用最广泛。针对多边形表面模型的主流变形方法有 Morphing 形状插值[12,39]、FFD[13,49]、骨骼变形[14,57]和基于物理的变形等方法。

在上一章中已经说过，Morphing 技术大体可以分为二维图像 Morphing 技术、二维形状 Morphing 技术和三维图形 Morphing 技术。

三维图形 Morphing 是指将一个三维物体光滑连续的变换为另一个三维物体，三维 Morphing 比二维 Morphing 要复杂得多，三维 Morphing 得到的中间帧是物体的模型而不是图像，所以三维 Morphing 中，一旦得到中间帧物体序列，就可以使用不同的摄像机角度和光照条件来进行重新绘制。三维 Morphing 与二维 Morphing 的思路类似，也是首先根据对应特征指定一空间变换，达到几何对齐的目的，然后将两个得到的扭曲变形体进行混合。Sloan[41]使用线性径向基函数，在抽象插值空间中实现高效的模型插值计算，根据关节体和人脸的样本状态，插值产生其连续范围内的任意形态。Lewis[38]详细分析了形状插值和骨骼驱动的变形技术，并统一使用 pose 空间的映射来表达物体变形，用于面部动画和人体变形。Singh[51]提出基于控制曲线的变形方法。Sumner[60]使用逆运动学求解变形，通过顶点间的变形向量传递实现模型变形。

三维 FFD 自由体变形也称为空间变形，空间变形是指将单个几何对象的形状做某种扭曲、变形，使它变换到所需的形状，在变换过程中，几何对象的拓扑关系保持不变。与 Morphing 不同，空间变形更具有随意性，该方法不直接操作物体，而是将物体嵌入到一空间，然后通过变形所嵌的空间，从而变形其中的物体。进行 FFD 变形的基本步骤如下：

（1）确定物体的顶点（或控制顶点）在 FFD 参数空间中的位置，建立 FFD 块的局部坐标系；

（2）变形 FFD 块，移动 FFD 块的控制顶点，生成变形后的物体空间；

（3）根据先前求出的顶点 FFD 局部坐标，确定空间变形后所对应的物体顶点的位置。

Sederberg[8] 首次提出基于空间变换的 FFD 变形方法，将物体内嵌在一个规则空间中，通过空间变形实现对任意拓扑结构内嵌物体的变形。Mac Cracken[49] 使用点阵网格划分变形空间进行 FFD 变形。

骨骼变形根据骨骼框架划分模型，然后计算骨骼对模型顶点的影响因子，但它一般需要足够的变形实例或复杂的手动配置获得合适的权值，以实现满意的变形结果。Paul[58] 计算对每个骨骼关节点变形的主要成分影响，实现基于图形硬件的非线性皮肤变形。Mohr[59] 使用一系列的示例姿态配置骨骼模型，通过调整变形模型的参数实现快速变形。

近年来细节保持的模型变形技术越来越受到重视，该技术提供了一个高质量模型变形的途径。细节保持方法将原始模型分离为一个分辨率较低的基础模型和若干个高变化率的细节模型，仅需对基础模型进行变形便可保持模型的高频细节。Huang[85] 将几何模型映射到一个包围原模型的基础网格上进行变形，从而减小问题的规模、迭代复杂度和内存消耗。Shi[70] 提出一种细节保持的网格操作框架，通过优化骨骼和顶点权值加速变形计算。Alexa 在 2003 年[78] 基于差分坐标实现局部模型过渡和变形，在 2006 年[80] 对基于离散 Laplace 和 Poisson 坐标的网格模型编辑进行了探讨。2004 年，Sorkine[76] 和 Yu[77] 分别将 Laplacian 坐标和 Poisson 坐标用于表面网格模型编辑，为细节保持的模型变形提供了一种新的思路。2007 年 Siggraph 在论文中，Nealen[16] 和 Wu[81] 分别给出了基于素描的和基于骨骼的细节保持变形算法。Wu[81] 将旋转角分解为多个小角度旋转，实现骨骼模型的大角度旋转变形，并在变形中实现细节保持。Kevin[62] 使用示例模型建立简化模型，通过逆运动方法计算关节点位置，实现与分辨率无关的网格变形。Martin[43] 引入黎曼几何距离概念，将三角形网格模型看做是形状空间中的点，计算给定模型的等距变形结果，最后使用一个多分辨率框架求解形状插值，获得任意状态的变形结果。

与以往的模型变形方法相比，本章提出的方法主要不同之处在于：

（1）对于不同的模型使用相同的统一规则网格作为变形基础模型，避免基础网格的生成计算，并可扩展变形方法的使用范围；

（2）在细节保持的基础上又实现了表面积保持，同时利用这种特性提供了一种实现物体变形表面皱褶的方法；

（3）统一规则网格的使用，为三维模型变形算法提供了一种硬件加速的可能性。

## 4.2　基于统一基础模型的角色变形方法

在三维模型变形过程中，细节保持是减少细节丢失、提高结果真实度的主要方法，它通过模型细节的变形前分离和变形后合成，保持了模型中的高频细节。在细节保持的变形方法中，基础模型主要有两种获取方法：一种是通过模型简化获得，另一种是通过低通滤波获得。细节模型通过原模型与基础模型求差获得。

以往的各类变形方法，均需要针对特定几何模型计算变形基础模型，并设计不同的变形算法，大大的加大了变形算法的复杂度。经研究发现，基础模型在一般意义上就是一个整体形状类似于原始模型且模型上各顶点之间的变化率较小的模型，如果将模型简化操作更进一步，也就是说，将简化获得的基础模型继续简化，一直简化到模型变化很小，此时，模型简化的结果将是一个表面平滑、顶点分布较均匀的模型，对于网格模型来说，就是一个均匀网格。

基于此研究发现，本章提出一种直接使用规则网格模型代替基础模型，进行三维模型的变形计算的方法。规则网格模型与原始模型的相关性很小，由于可以对不同的三维模型使用相同的基础模型，在本书中将这种基础模型定义为统一基础模型。本节针对一个独立的变形区域，研究其快速变形方法，并在该变形区域内实现柔软物体变形过程中产生的皱褶等特征。

### 4.2.1　规则网格的统一基础模型构建

一个完整的物体模型变形，可以通过对每一段独立的变形区域进行变形，然后再对变形后的网格根据控制点的位置变换坐标连为一个整体，最终实现整个模型的变形。物体的最基本变形方式是弯曲变形，各种形状的变形都可以由不同朝向和不同程度的基本弯曲组成。变形中心线的曲率最大处是该变形形状的控制点，根据控制点可以将整个变形物体分割为若干变形区域，每个变形区域中含有一个变形控制节点，每个控制节点与两侧的控制节点一起可以决定一段圆弧，每一段圆弧定义一个基本弯曲变形。

每一个基本的弯曲变形有两个表征：一是弯曲朝向和弯曲程度，分别用两个参数来定义这两个表征特性，弯曲方向角表示弯曲朝向，它的值为当前弯曲所在的平面与 *XOZ* 平面的夹角，在同一个弯曲平面中向左弯和向右弯的弯曲方向角差异 180°；二是弯曲旋转角表示弯曲程度，它的值为圆柱体基础网格顶平面与底平面的夹角，该角度越大则弯曲程度越大。

本节针对一段独立的变形区域，根据其弯曲角度等形状参数，直接使用圆柱体网格作为基础模型，从而减少基础模型的获取计算步骤。图 4-1 所示为使用统一基础网格的基础模型构建示意图，统一基础模型的规则网格的半径取原模型的

半径平均值，原模型凸出的部分将置于网格外，而凹进的部分将被网格包围。图4-2 所示为原模型与统一基础模型网格的分视图，可以更清晰地看出原模型与统一基础模型网格的关系。

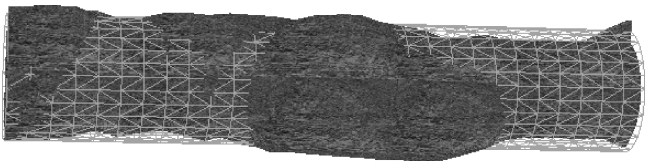
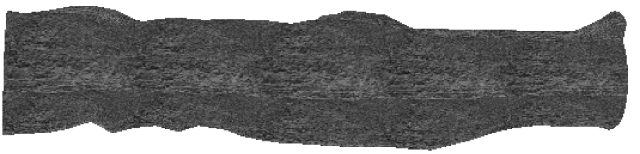
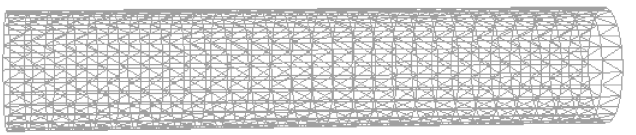


图 4-1　使用统一基础网格的基础模型构建示意图



a)



b)

图 4-2　原模型与统一基础模型网格的分视图

a）原模型　b）统一基础模型的网格

统一基础模型的构建方法为：对于每个独立变形区域，变形基础模型的规则圆柱体网格底面为当前变形区域与上一区域的接合面，底面半径为当前区域的横切面半径均值，高为区域长度。基础模型网格顶点在模型中水平方向按照旋转角度 $\theta$、垂直方向按照 $Z$ 坐标均匀分布。在三维模型的变形过程中，基础模型将通过弯曲旋转角度的改变或弯曲方向角度的改变实现变形。

直接使用规则网格模型作为基础模型的优点有如下几点：

（1）避免基础模型生成计算。基础模型一般通过模型简化或者低通滤波实现，需要大量的运算，在计算机图形学中，模型简化本身就是一个复杂问题，在细节保持的变形算法中，一般在预计算中进行基础网格的计算。根据原始模型的形状参数直接生成基础模型，避免了相当繁琐的基础模型生成步骤。

（2）提高基础模型的变形速度。一般的三维物体模型属于不规则网格，简化之后基础模型网格也是不规则网格。在不规则网格处理中，网格顶点之间的拓扑关系相当重要。不规则网格的变形，在处理模型顶点的时候，还需要考虑拓扑关系改变的处理，而规则网格的拓扑处理十分简单。

（3）规则网格有利于变形方法的硬件加速。不规则网格的顶点拓扑不确定性，造成其存储结构复杂，给模型处理带来很大的困难，同时也一直是一个困扰硬件加速的因素。通过使用规则网格的基础模型进行变形计算，有利于基础模型数据的快速存储与读取，也易于使用图形硬件加速变形的计算速度。

（4）规则网格的统一基础模型利于扩展变形算法的适用性。以往的各类变形方法，需要针对特定的几何模型计算变形基础模型，并设计不同的变形算法，通过直接使用形式统一的规则网格基础模型，使许多三维物体模型的基础模型变得十分相近，甚至相同，基础模型间的区别仅在于某些参数的不同，该算法在一定程度上提高了变形算法的通用性和可扩展性。

### 4.2.2　统一基础模型的变形计算

物体变形后的形状由控制曲线控制，控制曲线可以采用 NURBS、B 样条或其他任意曲线，但基本变形形式是弯曲变形，各种形式的变形可由基本弯曲变形组成，各种基本弯曲变形的局部区域可以进行独立计算，然后通过坐标系转换实现各区域的接合，获得整体的变形结果。为简化计算，以普通圆弧曲线作为基本弯曲变形的控制线，完整控制线上每三个相邻控制点定义一段圆弧控制曲线。3D 空间中的圆弧控制曲线可用其所在平面和圆弧角度两个参数来定义：一是控制曲线所在平面称为变形平面，它与 *XOZ* 平面的夹角称为变形方向角；二是圆弧角度称为变形旋转角，决定模型变形的弯曲程度大小，角度越大则弯曲程度越大，变形中表现为基础网格顶平面与底平面的夹角。

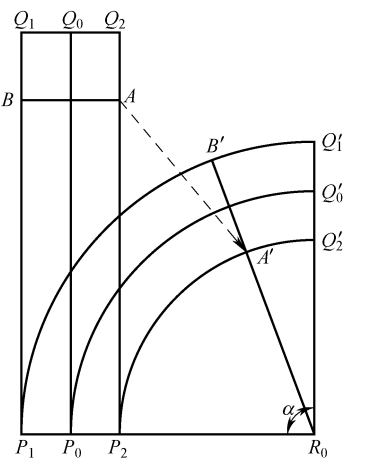


图 4-3　基础模型网格垂直截面图

图 4-3 所示为以圆弧作为控制曲线的基础模型网格垂直截面图。其中 $P_1P_0P_2$-$Q_1Q_0Q_2$ 是拉直状态的网格，$P_1P_0P_2$-$Q_1'Q_0'Q_2'$为弯曲状态的网格，$P_0Q_0$ 和 $P_0Q_0'$分别是两状态下的控制线，$\angle P_0R_0Q_0'$即 $\alpha$ 是变形旋转角。

柔性物体的表面可拉伸性很小，当设定拉伸系数为无穷大时，弯曲方向最外侧表面在变形前后长度保持不变，因此，圆弧半径 $|R_0P_0|$ 可由式（4-1）求得。

$$radius = \frac{|P_1Q_1'|}{\alpha} = \frac{|P_1Q_1|}{\alpha} = \frac{height}{\alpha} \tag{4-1}$$

式中，*height* 是基础模型拉直状态下的总高度；$\alpha$ 是该次弯曲变形的变形旋转角；由该半径和变形方向角即可得到焦点 $R_0$ 的位置坐标。

对于基础网格的任一顶点 *A*，首先计算该顶点所在的横截面，即图 4-3 中

$R_0A'B'$平面，然后根据该顶点在该横截面中的相对位置，计算出变形后的最终位置。

图 4-4 为基础模型的变形横截面示意图，其中 $R_0$ 为变形焦点，点 $O$ 为控制曲线与横截面的交点，点 $B$ 为顶点 $A$ 在变形平面 $R_0P_1$ 上的垂点，向量$\overrightarrow{AB}$垂直于变形平面，在弯曲变形中向量长度和方向保持不变，即顶点 $A$ 与投影点 $B$ 在变形前后的相对位置不变，因而，顶点 $A$ 变形后的位置可以通过向量和（$\overrightarrow{R_0B}+\overrightarrow{BA}$）得到。

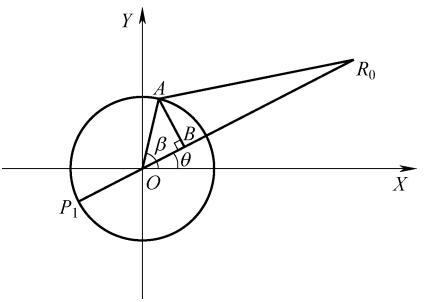



图 4-4　基础模型变形的横切面图

在变形过程中，基础模型网格以焦点 $R_0$ 为中心弯曲，垂点 $B$ 在变形平面中处于一条与变形控制曲线同圆心的弧线上，该弧线的半径可用式（4-2）求得，即

$$\begin{aligned} R_0B &= |R_0O| - |BO| \\ &= |R_0O| - radius * \cos(\beta-\theta) \end{aligned} \tag{4-2}$$

式中，*radius* 为基础模型的圆柱体底面半径；$\beta$ 为顶点 $A$ 的坐标方向角；$\theta$ 为变形弯曲方向角。

向量 $R_0B$ 的方向角等于本次弯曲变形的方向角，由于 $A$ 点与 $B$ 点在同一俯仰平面上，因此俯仰角 $\alpha_B=\alpha_A=\alpha*z_A/height$ 为顶点所在高度与总高度的比值。根据该向量的模、方向角和俯仰角，可以通过极坐标与普通坐标的转换求得控制点 $O$ 的位置坐标。

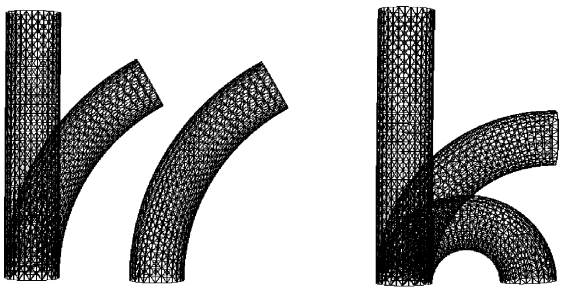
向量$\overrightarrow{AB}$在横截面的局部坐标系中，方向角为变形的弯曲方向角 $\theta$，长度为顶点 $A$ 到变形平面 $R_0P_1$ 的距离，因此向量$\overrightarrow{AB}$可用式（4-3）求得。其中，方向角 $\theta$ 为 $\angle XOR_0$。

$$\begin{bmatrix} x_{AB} \\ y_{AB} \end{bmatrix} = \left( \begin{bmatrix} x_A \\ y_A \end{bmatrix}, \begin{bmatrix} \sin(\theta) \\ \cos(\theta) \end{bmatrix} \right) * \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix} \tag{4-3}$$

针对变形基础模型中的所有顶点 $A_i$，计算其在变形平面中的垂点 $B_i$ 及其垂线向量$\overrightarrow{A_iB_i}$，通过求解向量和（$\overrightarrow{R_0B_i}+\overrightarrow{B_iA_i}$）得到顶点的初始弯曲变形后的位置。

图 4-5 所示为统一基础模型的变形结果示意图，图 4-5a 所示为以 60°角度弯

曲变形后的基础模型网格状态，图 4-5b 所示为以 90°和 180°角度弯曲变形后的网格状态。



a)　　b)

图 4-5　统一基础模型的变形结果

a）60°角度弯曲变形后的基础模型网格状态　b）90°和 180°角度弯曲变形后的网格状态

### 4.2.3　统一基础模型变形的表面积保持

细节是物体模型的一种表征相邻顶点之间变化梯度的属性，细节保持可以看做是模型的一种属性保持手段，对提高变形结果真实感具有重要的作用。表面积是柔软物体和壳体的另一大特征属性，但目前的变形方法对它关注不足。柔软物体如软体动物、橡胶以及毛绒玩具等，它们的一个显著特征是表面水平可拉伸性低但垂直可挤压性高，总面积总试图保持不变，弯曲方向最外侧在变形过程中不会被拉伸，而在弯曲方向内侧受挤压部分会出现多条凹凸纹皱褶。

目前的变形方法大多针对刚体变形，而针对柔软物体或可弯曲物体的变形研究较少。实现柔软物体的变形一般使用基于物理的方法，但这种方法需要大量的计算，大多无法达到实时变形。

本章提出的统一基础模型，由于规则网格的易处理性，可以在实时处理的前提下，仿真柔软物体的表面变形皱褶，同时实现变形过程中的表面积保持。与基于物理的变形方法相比，基于统一基础模型的柔软物体变形有如下优点：

（1）变形算法简单、计算速度快，可以实时地产生能接受的变形结果；

（2）物体的柔软度是可控的，可以通过调整变形参数，改变皱褶的间隔和皱褶数量，变形模型会自动调节皱褶的形状，保持表面积不变。

以下是基于统一基础模型的表面积保持原理及其实现方法。

统一基础模型的规则网格顶点可以划分为若干列，每一列顶点在本书中称之为顶点列。弯曲变形除弯曲最外侧顶点列外，其余各顶点列的表面均被挤压。各列在弯曲与拉直状态的列长度比为压缩系数，该系数越小表示压缩程度越大，压缩系数最大为 1 表示未被压缩。该系数表征基础模型的变化率，在变形过程中，

将据此计算阻尼振荡曲线函数，实现基础模型的表面积保持。

在统一基础模型变形中，是使用阻尼振荡曲线叠加到模型表面来实现表面积保持的，叠加过程需首先计算基础模型在弯曲变形过程中的表面积受压缩系数，根据该系数计算阻尼振荡曲线函数，使曲线的总长度等于拉直状态下的顶点列长度，从而实现模型变形中的表面积保持，并用曲线的形状模拟柔软物体弯曲变形内侧的皱褶。

在基础模型网格垂直截面图 4-3 中，最外侧顶点列 $P_1Q_1$ 弯曲变为 $P_1Q_1'$，该列在变形中未被压缩即 $|P_1Q_1| = |P_1Q_1'|$，而对于其他列上的任一顶点 $A$，其所在列的受压缩系数可推导为式（4-4），即顶点列的圆弧半径与最外侧圆弧半径之比。

$$Scale_A = \frac{|P_2Q_2'|}{|P_2Q_2|} = \frac{|P_2Q_2'|}{|P_1Q_1|} = \frac{|P_2Q_2'|}{|P_1Q_1'|} = \frac{R_A}{R_1} \tag{4-4}$$

为继续求解式（4-4）获得两圆弧半径之比，由基础模型变形的横切面图 4-4 中可以得出，$B$ 为 $A$ 在变形平面上的垂点，在变形平面中弯曲变形时点 $A$ 与 $B$ 的弯曲圆弧半径相同，因此顶点 $A$ 的所在圆弧半径可通过点 $B$ 的圆弧半径求得，顶点 $A$ 的基础模型弯曲压缩系数可由式（4-5）求得，其中 $rad$ 为基础模型网格的底面半径。

$$Scale_A = \frac{R_A}{R_1} = \frac{R_B}{R_1} = \frac{|OR_0| - rad\cos(\beta - \theta)}{|R_0P_1|} \tag{4-5}$$

在基础模型网格变形过程中，最外侧顶点列 $P_1Q_1$ 长度不变，其他所有顶点列的长度都会被压缩，造成表面积减少。为实现变形表面积保持，引入新的皱褶仿真函数。经研究发现，表面皱褶在变形区域中中间部分起伏最大，向两端逐渐减弱，该特征与振幅递减的阻尼振荡曲线相同，可以采用该曲线仿真物体表面的皱褶，并根据模型表面各处压缩系数计算曲线参数，获得表面积保持的基础模型网格。阻尼振荡曲线的函数定义见式（4-6）。

$$f(x) = A\left(1 - \frac{x}{x_{\max}}\right)\cos\left(\frac{2k\pi}{x_{\max}}x\right) \tag{4-6}$$

式中，$A$ 是振荡曲线的最大振幅；$x_{\max}$是振荡曲线的最大振荡时间；$k$ 是振荡周期数，$k$ 越大则振荡次数多，在变形仿真中是用户设定的形状因子，决定皱褶个数。

通过保持各顶点列在变形前后的总长度不变，实现基础模型变形前后的表面积保持不变，也就是说，通过振荡曲线调整模型顶点位置，使顶点列弯曲状态下的总长度等于拉直状态的总长度。基础模型的每一顶点列共用一条阻尼振荡曲线，曲线从中间向两端对称延伸，其长度可用式（4-7）计算。

$$length = 2\int_0^{X_{\max}} \sqrt{1 + f'(x)^2}\,dx \tag{4-7}$$

式中，$f'(x)$是阻尼振荡曲线函数的微分；$X_{\max}$是最大振荡时间，即该顶点列的圆

弧半长度 $X_{\max} = height \cdot Scale/2$。

最大振幅 $A$ 是调节振荡曲线总长度的主要参数，根据顶点列在变形前后的总长度不变的条件即 $length = height$，可求得参数 $A$。由于振荡曲线函数的微分形式复杂，积分困难，本节采用将整个压缩系数空间按 0.01 间隔划分，采用数值方法求得每个压缩系数所对应的阻尼曲线最大振幅 $A$。

在基础模型表面积保持计算中，根据各顶点在弯曲圆弧上的位置，通过阻尼振荡曲线函数计算该顶点的函数值，该值即为该顶点为仿真皱褶所需要的高度调整值。依次计算各列顶点，求得变形基础模型上所有顶点的高度调整值，然后根据该值调节基础模型网格顶点的位置，调节方法是，对于每个顶点，从其变形横截面上的控制曲线交点，向该顶点发射一条射线，沿该射线方向叠加高度调节值，计算顶点的新位置。调节之后，变形基础模型的各顶点列在变形过程中可保持长度不变，从而实现整个基础模型在变形中保持表面积不变。

非表面积保持与表面积保持的基础模型网格在 90°弯曲状态下的比较如图 4-6 所示。从变形网格状态中可以看出，表面积保持的变形更有一种受挤压感。从观察者的角度，表面积保持的变形更能体现出这是一个从其他状态变化得到的结果，而不是它的初始状态就是如此。

图 4-7 所示为基础模型在其他弯曲角度下的网格状态，从图中可以看出，受挤压的程度越大，表面皱褶越明显，皱褶也越尖锐，这也是柔软物体的表面积保持特性在受挤压时的实际效果。

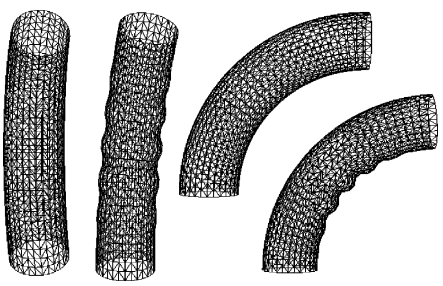


图 4-6　表面积保持的基础模型网格 1

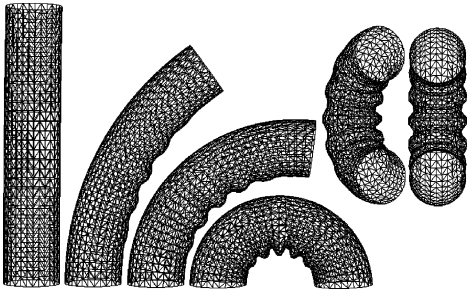


图 4-7　表面积保持的基础模型网格 2

### 4.2.4　基于统一基础模型的细节保持变形

细节保持是减小变形失真提高变形逼真度的重要方法，它通过分离原几何模型为基础模型和细节模型，在变形中只处理基础模型，变形后将细节模型再合并到基础模型上，细节模型不参与计算从而减少细节丢失。

广义的模型细节，是指模型表面或内部各顶点之间的变化率，如 LOD 算法中将模型细节分为粗细不同的层次模型[17]。狭义的模型细节，是特指梯度等参

数变化率较大的部分，即高频部分。基础模型一般分辨率较低，顶点间的变化率小，细节模型包含高变化率的模型细节，是原模型网格与基础模型的差。大部分的模型处理或变形过程常常会使模型丢失尖角凸起等细节，模型表面变得平缓。细节保持的变形方法将原始模型分离为一个分辨率较低的基础模型和若干个高变化率的细节模型，在变形过程中，只对基础模型进行变形，变形后再将细节模型合成到变形后的基础模型上，从而保持模型上的高频细节。

细节保持方法中的细节模型，需要通过原模型与基础模型进行径向求差运算获得，每个顶点表征的是原模型与基础模型的差。为更好地保持模型细节，使用的细节模型与原始模型拓扑相同，两模型顶点一一对应。统一基础模型是规则网格，而原始模型一般是不规则网格，在径向求差的过程中，使用三线性采样，获取基础模型的径向值。变形基础模型之后，再将该细节模型合成到基础模型上，实现细节保持的模型变形。

图 4-8 所示为细节模型计算的纵切面图，其中 $R_0$ 是变形焦点，$P_0Q_0'$为变形控制曲线，平面 $R_0P_0Q_0'$为变形平面，$P_1P_2Q_1'Q_2'$是原模型对应的基础模型，不规则曲线 $M$ 和 $N$ 是原物体模型的交线。

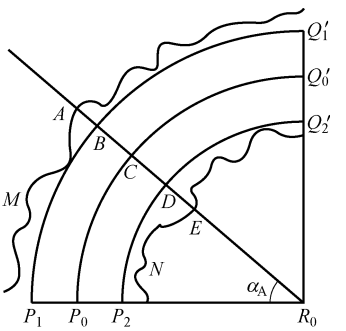



图 4-8 细节模型计算的纵切面图

控制线交点 C 可以通过向量$|\overrightarrow{R_0P_0}|$旋转获得，旋转后的向量$|\overrightarrow{R_0C}|$的旋转角等于变形方向角，俯仰角 $\alpha_A$ 为顶点 $A$ 变形旋转角，可以根据式（4-8）求得。

$$\alpha_A = \arctan \frac{z_A - z_O}{\sqrt{dx^2 + dy^2}} \tag{4-8}$$

以模型顶点 $A$ 为例，其模型细节的获取方法是，作一过顶点 $A$ 的横切面，$R_0A$为该切面在变形平面中的投影，该切面与原模型以及基础模型相交获得的横切面如图 4-9 所示，其中 $C$ 为控制曲线与切面的交点，圆 $BQD$ 为基础模型与切面的交线，$\overrightarrow{BA}$为顶点 $A$ 对应的模型细节值，即向量$\overrightarrow{CA}$与$\overrightarrow{CB}$的差。

针对原模型中的每一个顶点，通过该点与对应控制点的距离求得模型细节值，即 $Detail = |\overrightarrow{AC}| - rad$，其中 $rad$ 为基础模型的圆柱体底面半径。保存各顶点的旋转角 $\alpha_i$、横切面内方向角 $\theta_i$ 以及细节值，得到变形所需的细节模型。

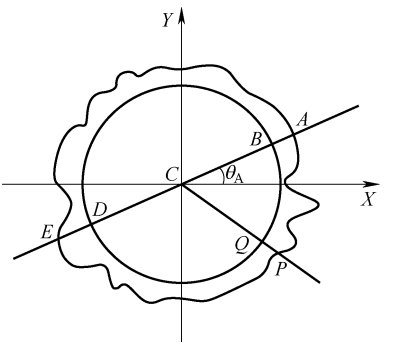



图 4-9 细节模型计算的横切面示意图

在变形过程中，对基础模型进行变形计算，然后将细节模型合成到基础模型上，以还原原模型的细节。细节模型的合成实际上是其获取过程的反向运算，对于细节模型的任一顶点 $A$，根据其 $\alpha$ 和 $\theta$ 值求得其在基础模型上的投影点 $B$。

细节模型与基础模型的合成方法是，采用三线性插值法获得基础模型投影点 $B$ 的坐标，从而得到向量$\overrightarrow{CB}$，将顶点 $A$ 的模型细节值与向量$\overrightarrow{CB}$叠加求得向量$\overrightarrow{CA}$，根据$\overrightarrow{CA}$和 $C$ 点位置求得顶点 $A$ 的变形后位置。针对细节模型的每一顶点，均采样其对应的基础网格模型坐标，然后叠加模型细节值，即可获得整个模型的最终变形结果。

## 4.3　基于统一基础模型的三维角色变形算法实现

本章针对三维物体的变形问题，提出一种基于统一基础模型的变形方法，对于不同的物体模型，使用统一的基础模型实现变形计算，并使用阻尼振荡函数调节模型表面，实现表面积保持，同时仿真柔软物体的弯曲皱褶。

模型变形的整个过程可划分初始化部分和变形计算部分，主要过程如下：

（1）初始化部分。首先，根据需变形物体的长宽高等基本参数，建立规则网格的基础模型；然后，变形基础模型，使其弯曲角度与模型控制点的旋转角相同，使用径向计算进行原始模型的细节分离，将原模型向基础模型径向求差获得其细节模型；最后，计算各压缩系数对应的阻尼曲线最大振幅。

（2）变形计算部分。首先，计算变形物体各控制点的当前旋转角，变形基础模型至新的姿态；然后，根据旋转角计算顶点列的压缩系数，获取阻尼振荡曲线的最大振幅，根据阻尼曲线函数计算高度调节值，获得表面积保持的基础模型；最后，将细节模型合并到变形后的基础模型上，获得几何模型的变形结果，渲染该变形结果。

## 4.4　三维角色变形关键代码

与上一章的二维角色变形相比，本章的三维角色变形不同代码主要是变形计算部分。本章的三维角色变形计算包括主网格采样、几何控制网格变形、面积保持计算、覆盖网格合成和细节网格合成几部分。下面依次是这几部分的关键代码。

### 4.4.1　主网格采样算法

主网格采样算法的作用是计算采样坐标，然后根据当前列角度和弯曲方向角，返回原始网格在变形网格上的纵坐标。计算采样坐标代码如下：

```
void CalSampleCrd(float xcrd,float ycrd,float zcrd)
{
    //首先根据实际主网格 XY 坐标，计算其旋转角，再减去弯曲角
    //得出覆盖 X 坐标，然后根据 Z 坐标，比较总高度，得出覆盖 Y 坐标
    float thet = atanf (ycrd/xcrd);
    thet < 0? thet + = PI:0;        //此时 thet = (0,pi)
    ycrd < 0? thet + = PI:0;        //此时 thet = (0,2* pi)
    thet - = bendDirAngle;
    while(thet < 0) thet + = 2* PI;
    while(thet > 2* PI) thet - = 2* PI;
    //对折
    thet > = PI? thet = 2* PI - thet: 0;
    sampleCrdX = thet/PI;
    ASSERT((sampleCrdX > = 0) &&(sampleCrdX < = 1));
    //垂直方向中间是弯折中心
    sampleCrdY = zcrd/height* 2 - 1;
    sampleCrdY < 0? sampleCrdY = - sampleCrdY: 0;
    //将 Z 采样坐标放大一倍，即可得到向中间紧凑的效果
    sampleCrdY * = 1.2;
    sampleCrdY > 1? sampleCrdY = 1: 0;
    ASSERT((sampleCrdY > = 0) &&(sampleCrdY < = 1));
    sampleCrdX * = (float)(vNumRow - 2)/vNumRow;
    sampleCrdY * = (float)(vNumCol - 2)/vNumCol;
}
```

获得采样坐标后，根据覆盖网格，插值得到原始网格的细节采样值，代码如下：

```
float SampleAltitude()
{
    int xCovCrd = (int)(sampleCrdX* vNumRow);
    int yCovCrd = (int)(sampleCrdY* (vNumCol - 1));
    float offsetX = sampleCrdX* vNumRow - xCovCrd;
    float offsetY = sampleCrdY* vNumCol - yCovCrd;
    //采样点落在一个正方形中，三线性插值
    float * pLeft = mUniCovDemArray + xCovCrd* vNumRow + yCovCrd;
    float * pRight = pLeft + vNumRow;
```

```
    float valueA = (* pLeft)* offsetX + (* pRight)* (1 - offsetX);
    pLeft + +; pRight + +;
    float valueB = (* pLeft)* offsetX + (* pRight)* (1 - offsetX);
    float valueC = valueA* offsetY + valueB*  (1 - offsetY);
    ASSERT (fabsf (valueC) < radius);
    return valueC;
}
```

通过调用上面两个函数，可以采样整个模型的细节高程值，全模型的采样代码如下：

```
float* CalCovDem4PriMesh()
{
    //采样获得对应于主网格的覆盖高程值，返回数据数组指针
    int sizePCM = mPriNumXY* mPriNumZ;
    //if(mPriCovDemArray! =NULL) delete[]mPriCovDemArray;
    if(mPriCovDemArray ==NULL) mPriCovDemArray =new float[sizePCM];
    ASSERT(mPriCovDemArray! =NULL);
    memset(mPriCovDemArray,0,sizePCM* sizeof(float));
    //针对每一个顶点，计算其浮动值
    float * ppca = mPriCovDemArray;
    mVertex * ppma = mPriMeshArray;
    for(int i =0; i < sizePCM; i + +)
    {
        //每列两头的点不要参与计算
        //计算采样坐标
        CalSampleCrd(ppma -> x,ppma -> y,ppma -> z);
        //采样浮动值
        * ppca = SampleAltitude ();
        ppca + +; ppma + +;
    }
    return mPriCovDemArray;
}
```

### 4.4.2　计算控制网格的放缩系数

在三维变形过程中，实现表面积保持是本章所提出算法的主要特色，而表面积保持网格需要根据控制网格在变形过程中所产生的放缩系数进行求解。控制网

格的每一列顶点列的放缩系数，由它切割轴向连线得到的距离比例决定，求解代码如下：

```
bool CalLenScale()
{
    if(lenScale == NULL) lenScale = new float[vNumXY];
    ASSERT(lenScale! = NULL);
    //总长度 = length(rotCenter,axis0) + radius
    float ratioT = -1/lenCenter/(lenCenter + radius);
    mVertex * pmv = mVexArray;
    float vectX,vectY,vectLen;
    float * pls = lenScale;
    for(int i = 0; i < vNumXY; i + + )
    {
        //对于每一列顶点,计算其放缩系数
        vectX = pmv -> x - dirCenterX;
        vectY = pmv -> y - dirCenterY;
        vectLen = sqrtf(vectX* vectX + vectY* vectY);
       * (pls + + ) = (vectX* dirCenterX + vectY* dirCenterY)* ratioT;
        pmv + = vNumZ;
    }
    return true;
}
```

### 4.4.3 面积保持网格的计算

本文模型变形过程中的面积保持，通过在控制网格上叠加面积保持网格实现，面积保持网格的原理是使每一列顶点都产生一个曲线形状，从而保持该顶点列总长度。计算顶点列面积保持曲线振幅的代码如下：

```
float CalMaxSwing(float lenScaleArray,float* tempArr2,int sizeArr)
{
    float xMax = cycNum* 2* PI;
    float * pta = tempArr2;
    int i; float j = 0,jIvl = 1.0f/uniCovNum;
    float lsls = lenScaleArray* xMax;
    lsls = lenScaleArray* lenScaleArray* jIvl* jIvl;
    float Aleft = 0,Aright = 1,Amiddle,AA;
```

```
    float cosLenL = 0,cosLenR = 0,cosLenM = 0;
    //先求初始值，使用 Aleft，Aright 求 len，Aleft == 0 时不需要积分
    cosLenL = lenScaleArray* jIvl* sizeArr;
    pta = tempArr2;
    AA = Aright* Aright;
    for(i = 0; i < sizeArr; i + + )
    {
        cosLenR + = sqrtf(lsls + (* (pta + + ))* AA);
    }
    //然后使用迭代法，求解振幅 A
    while(fabsf(Aright - Aleft) > 0.01)
    {
        //在 Aleft 和 Aright 之间求一个线性中间值 Amiddle
        //根据中间值结果，调整 Aleft 和 Aright 的新值
        Amiddle = Aleft + (Aright - Aleft)* (xMax - cosLenL)/
                  (cosLenR - cosLenL);
        AA = Amiddle* Amiddle;
        pta = tempArr2; cosLenM = 0;
        for(i = 0; i < sizeArr; i + + )
        {
            cosLenM + = sqrtf(lsls + (* (pta + + ))* AA);
        }
        if(fabsf(cosLenM - xMax) < 0.05) break;
        if(fabsf(cosLenM - cosLenL) < 0.05) break;
        if(fabsf(cosLenM - cosLenR) < 0.05) break;
        if(cosLenM > xMax)
        {
            Aright = Amiddle; cosLenR = cosLenM;
        }else{
            Aleft = Amiddle; cosLenL = cosLenM;
        }
    }
    return Amiddle;
}
```

面积保持网格曲线的最大振幅求出后，将根据该振幅求得整个面积保持网

格，面积保持网格的位移量保存在 mUniCovDemArray 数组中，使用没有频率调整的变振幅余弦曲线计算面积保持网格的代码如下：

```
bool CalUniformCovRetDEM()
{
    float xMax = cycNum* 2* PI;
    int sizeArr = vNumRow* vNumCol;
    if(mUniCovDemArray ==NULL) mUniCovDemArray =new float[sizeArr];
    ASSERT(mUniCovDemArray! =NULL);
    //有了振幅，可以得出方程式，然后根据方程式计算 Y 值即可
    //先列后行，同一列的挨在一起，每列 vNumRow 个元素
    //对于每一列，x =0...xMax,
    int i, j;
    float xRowIvl = xMax/(vNumRow -1);
    float * pls = lenScaleArray;
    float * pmca = mUniCovDemArray;
    int inxSwing; float swingL,swingR,swing;
    float xRowValue,tempFloat;
    for(i =0; i <vNumCol; i + +)
    {
        ASSERT((* pls) >0.3);//小于 0.3 的效果将变差
        //swing 改为取用并线性插值
        //maxSwingArray 是逆序存储的，第一个 lenScaleArray ==1
        //去尾法取整
        inxSwing = uniCovNum - (int)((* pls)* uniCovNum);
        swingL =* (maxSwingArray + inxSwing);
        swingR =* (maxSwingArray + inxSwing -1);
        swing = swingL + (((* pls)* uniCovNum) - (uniCovNum -
                inxSwing)) *(swingR - swingL);
        tempFloat = swing/xMax;
        //求函数值，按无放缩的方程求 Y 值
        // (X 与放缩有关，Y 与放缩无关)
        xRowValue =0;
        for(j =0;j <vNumRow;j + +)
        {
           * pmca = (swing - tempFloat* xRowValue)* cosf(xRowValue);
```

```
            ASSERT(* pmca < 10);
            xRowValue + = xRowIvl;
            pmca + + ;
        }
        pls + + ;
    }
        return true;
}
```

### 4.4.4　各部分网格合并算法

整个三维模型变形过程是：首先进行控制网格的变形；然后在每次的控制网格变形之后，计算其相应的面积保持网格；最后将面积保持网格和细节网格一起合并到控制网格上，实现三维角色变形部分的快速变形。各部分网格合并部分的代码如下：

```
bool SynthCoverMesh ()
{
    float maxHeight = 1.5* height;
    //就是根据原来的法向量方向，扩展 Cover 那么长，从而得出新的坐标
    mVertex * ppmd = meshDeformed;
    mVertex * pmva = mVexArray;
    float * ppca = mPriCovArray;
    float rotThet,axisLen,axisLenNew;
    float focusX,focusY,focusZ;
    float vectX,vectY,vectZ;
    float ratioAH,cosV1;
    ratioAH = bendRotAngle/height;
    for(int i = 0; i < sizePriMesh; i + + )
    {
        //首先根据 z 计算出当前的中心点坐标
        //heightCur/height* bendRotAngle 当前面内转角
        rotThet = pmva -> z* ratioAH;
        //中心轴坐标
        cosV1 = 1 - cosf(rotThet);
        focusX = dirCenterX* cosV1;
        focusY = dirCenterY* cosV1;
```

```
            focusZ = lenCenter* sinf(rotThet);
            //然后根据中心点至当前点的向量,求长度
            vectX = ppmd -> x - focusX;
            vectY = ppmd -> y - focusY;
            vectZ = ppmd -> z - focusZ;
            axisLen = sqrtf(vectX* vectX + vectY* vectY + vectZ* vectZ);
            //然后放缩长度，并重新求新坐标
            axisLenNew = (axisLen + (* ppca)) /axisLen;
            ppmd -> x = focusX + vectX* axisLenNew;
            ppmd -> y = focusY + vectY* axisLenNew;
            ppmd -> z = focusZ + vectZ* axisLenNew;
            ASSERT (ppmd -> z < maxHeight);
            ppca + +; pmva + +; ppmd + +;
        }
        //最终覆盖网格更新到变形后的主网格上
        return true;
    }
```

## 4.5 实验结果与分析

基于本章的三维角色变形思想与方法，现在已经在 PC 上实现了一个基于统一基础模型的三维角色变形演示程序。本章实验的硬件配置为 Intel P4 3.0G CPU，1GB 内存，nVidia GeForce FX6600 128M 显卡，软件环境为 Windows XP 操作系统，编程环境为 Microsoft Visual Studio 2005，3D 开发环境为 OpenGL。

本节的实验使用的模型，一个是一段通过程序生成的数字纹理的三维桶装体，另一个是一棵树木主干的三维模型。在实验过程中，分别对这两个三维模型实现变形，通过实验结果及数据的分析，验证本章提出的三维模型变形方法的有效性。

本节的实验主要包含如下几部分：

（1）验证基于统一基础模型的变形技术的可行性和有效性。本章提出一种规则网格的变形基础模型，使用阻尼振荡曲线实现物体表面的皱褶起伏，仿真弯曲变形内侧的皱褶，实现变形过程中的模型细节保持。通过给出三维模型的变形结果，说明该方法的可行性和有效性。

（2）获得变形计算时间与基础模型的网格精细度的关系。本章提出了针对三维角色模型变形的统一基础模型，该模型与变形目标对象相关性很小，通过规则

网格的易处理性和可硬件加速特性，提高三维模型变形的处理速度。通过分别统计不同网格精细度下的计算时间，获得基础模型精细度对变形计算时间的影响。

（3）获得变形计算时间与原始模型的顶点数目的关系。本章提出的基于统一基础模型的变形方法，基础模型与目标对象的相关性很小。通过选择不同细节层次的原始模型，统计模型变形的各步骤运行时间，获得原始模型对变形计算时间的影响。

### 4.5.1　基于统一基础模型的三维角色变形结果

统一基础模型可以通过叠加阻尼函数曲线，实现表面积保持。因此，基于统一基础模型的变形可以选择性实现表面积保持，也可以跳过该步骤获得非表面积保持的变形结果。图 4-10 为基于统一基础模型的非表面积保持的变形结果，从图中可以看出，基于统一基础网格的变形方法可以实现各种角度下的模型变形。由于在变形过程中实现了细节保持手段，因此变形结果表现出良好的真实感。

a)　　b)　　c)

图 4-10　基于统一基础模型的非表面积保持变形结果

a）60°弯曲变形　b）90°弯曲变形　c）180°弯曲变形

图 4-11 所示为三维模型的非表面积保持和表面积保持的变形结果比较，其中图 4-11a 所示为 60°弯曲的结果比较，图 4-11b 所示为 90°弯曲的结果比较。

a)　　b)

图 4-11　非表面积保持与表面积保持的变形结果比较

a）60°弯曲的结果比较　b）90°弯曲的结果比较

从图 4-11 的变形结果比较中可以看出，表面积保持的变形更有一种受挤压

感，从观察者的角度，表面积保持的变形结果一方面能体现出这是一个从其他状态弯曲变化而来的状态，而不是它的初始状态就如此。另一方面，表面积保持的变形结果中物体表面的纹理失真小，而非表面积保持的变形结果中，弯曲内侧的纹理发生明显收缩，这是由于纹理坐标是基于模型表面计算的，因为表面积保持不变，所以不需要重新采样纹理，从而避免了纹理失真。

本章的变形方法可以针对不同的物体柔软度，通过调节覆盖阻尼曲线参数改变皱褶的数量。如图 4-12 不同皱褶数量的变形结果比较所示，其中图 4-12a 所示为 60°弯曲状态下皱褶数量分别为 2、3、5 的变形结果比较，图 4-12b 和图 4-12c 分别是 90°和 180°弯曲状态下皱褶数量分别为 2、3、5 的变形结果比较。

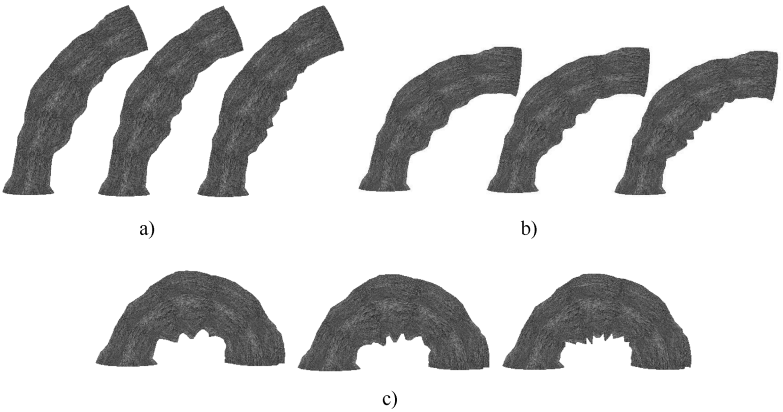



图 4-12　不同皱褶数量的变形结果比较

a）60°弯曲变形结果比较　b）90°弯曲的变形结果比较　c）180°弯曲的变形结果比较

图 4-13 表面积保持的模型变形结果 1 和图 4-14 表面积保持的模型变形结果 2 为三维模型变形的结果截图，由运行结果可以看出，该方法可以有效仿真三维物体的变形结果，使用皱褶实现变形过程中的表面积保持，同时能够很好地保持模型的原有细节。

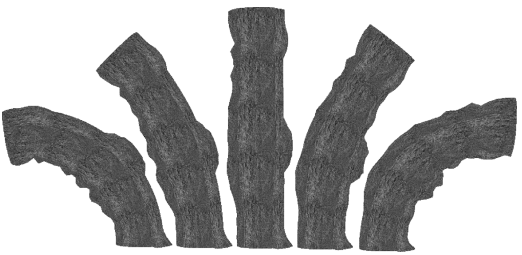


图 4-13　表面积保持的模型变形结果 1

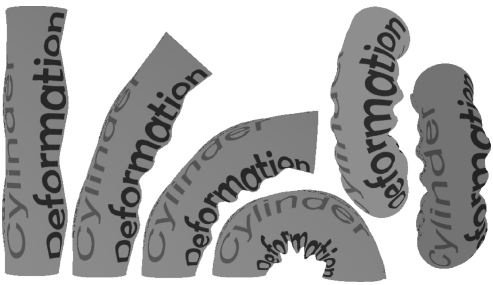



图 4-14　表面积保持的模型变形结果 2

## 4.5.2　变形计算时间与基础模型精细度的关系

本节实验针对基础模型的变形步骤，改变模型网格精细度，统计各步骤的执行时间，每一组统计数据的各步骤时间来自于 100 帧变形时间的平均值。实验所获得的不同网格精细度下的基础模型变形时间见表 4-1，实验数据表明，对于 1 万顶点规模的基础模型，每帧变形计算总时间可以保持在 30ms 左右，能够达到实时变形计算与渲染要求。

**表 4-1　不同网格精细度下的基础模型变形时间**　　（单位：ms）

| 基础模型网格顶点数 | 基础模型弯曲变形 | 表面积保持 |
|---|---|---|
| 5 × 5 | 0. 03232 | 2. 419 |
| 10 × 10 | 0. 09570 | 2. 763 |
| 15 × 15 | 0. 1861 | 2. 890 |
| 20 × 20 | 0. 3737 | 3. 711 |
| 25 × 25 | 0. 4716 | 4. 084 |
| 30 × 30 | 0. 7265 | 4. 383 |
| 35 × 35 | 0. 8980 | 4. 512 |
| 40 × 40 | 1. 184 | 5. 227 |
| 45 × 45 | 1. 433 | 6. 082 |
| 50 × 50 | 1. 914 | 6. 752 |
| 55 × 55 | 2. 213 | 7. 486 |
| 60 × 60 | 2. 819 | 8. 786 |
| 70 × 70 | 3. 613 | 10. 70 |
| 80 × 80 | 5. 435 | 14. 61 |
| 90 × 90 | 6. 318 | 16. 15 |
| 100 × 100 | 7. 866 | 20. 24 |

（续）

| 基础模型网格顶点数 | 基础模型弯曲变形 | 表面积保持 |
|---|---|---|
| 120 × 120 | 10. 62 | 26. 37 |
| 140 × 140 | 14. 49 | 34. 86 |
| 160 × 160 | 18. 45 | 45. 78 |
| 180 × 180 | 22. 98 | 57. 35 |
| 200 × 200 | 28. 50 | 69. 64 |

根据表 4-1 中的数据所绘制的基础模型变形时间与其精细度的关系如图 4-15 所示，从图中可以看出，随着网格精细度的增加，表面积保持步骤的增长速度比基础模型弯曲变形速度要快，这也说明表面积保持过程的计算量比基础模型弯曲变形更大。

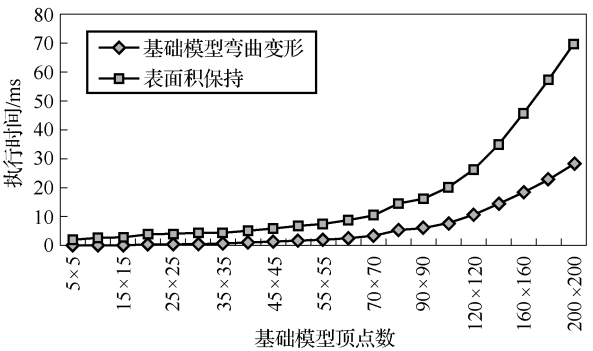



图 4-15　基础模型变形时间与网格精细度关系图

## 4. 5. 3　变形计算时间与原始模型顶点数的关系

本节实验使用图 4-14 中给出的一段三维变形模型，使用 30 个三角形的基础模型进行变形，通过 LOD 细化和简化，选择不同细节层次的原始模型，统计模型变形的各步骤运行时间，所得的运行时间数据见表 4-2。

**表 4-2　不同精细度原始模型的运行时间**　（单位：ms）

| 基础网格顶点数 | 基础模型变形 | 细节模型合成 |
|---|---|---|
| 10 × 10 | 5. 541 | 1. 577 |
| 15 × 15 | 5. 706 | 1. 803 |
| 20 × 20 | 5. 791 | 1. 828 |
| 25 × 25 | 5. 920 | 2. 029 |
| 30 × 30 | 5. 775 | 2. 022 |

（续）

| 基础网格顶点数 | 基础模型变形 | 细节模型合成 |
|---|---|---|
| 35 × 35 | 5. 861 | 2. 304 |
| 40 × 40 | 5. 900 | 2. 396 |
| 45 × 45 | 5. 678 | 2. 469 |
| 50 × 50 | 5. 907 | 2. 711 |
| 60 × 60 | 5. 789 | 3. 019 |
| 70 × 70 | 5. 791 | 3. 327 |
| 80 × 80 | 5. 607 | 3. 631 |
| 90 × 90 | 5. 713 | 3. 757 |
| 100 × 100 | 5. 728 | 4. 887 |
| 120 × 120 | 5. 774 | 5. 650 |
| 140 × 140 | 5. 825 | 7. 353 |
| 160 × 160 | 5. 757 | 10. 16 |
| 180 × 180 | 5. 744 | 12. 17 |
| 200 × 200 | 5. 830 | 12. 79 |

根据表 4-2 中的数据，所得出的模型变形时间与原始模型顶点数的关系如图 4-16 所示，图中显示，基础模型的变形时间基本保持不变，细节合成时间随原始模型顶点数的增加而增加。

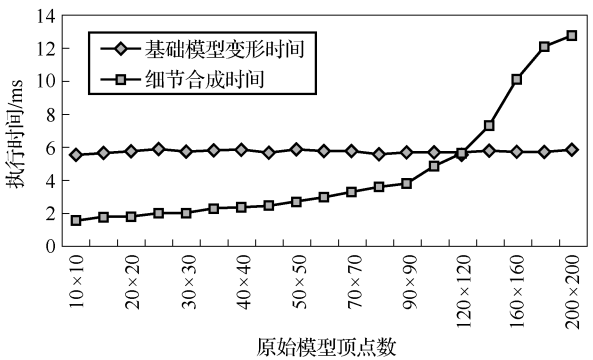



图 4-16　模型变形时间与原始模型顶点数的关系

在本节的实验中发现，在基础模型不变的情况下，由于基础模型的变形时间不变，三维模型的变形时间随原始模型顶点数增长而增长较缓，但当基础模型与原始模型相差很大时，将造成变形结果出现不平滑感，这里建议基础模型顶点数与原始模型的顶点数之比应不小于 1 ∶ 3，所支持的实时变形的原始模型的顶点数

为 1 万顶点左右。

## 4.6 本章小结

本章针对三维物体模型变形问题，提出了一种基于统一基础模型的变形方法。本方法是局部变形方法：根据控制点划分独立的变形区域，每个区域通过控制点控制变形后的形状；对于每一局部变形区域，使用规则圆柱体网格作为基础模型，减少基础模型的生成计算步骤，并由此提出与原模型基本无关的基础模型变形算法；通过叠加阻尼振荡曲线保持基础模型的顶点列长度不变，从而实现表面积保持，并同时仿真物体受挤压产生的表面皱褶；在变形前后以径向计算进行细节获取和合成实现细节保持。

实验证明，本方法能够快速有效地仿真三维物体的变形，并在变形过程中通过表面积保持和细节保持，提高变形结果的真实感，对于 1 万个顶点内的 3D 模型，可以达到 30FPS 的变形渲染帧速率。

进一步的工作将进行基于复杂控制曲线的变形技术研究，如 Bezier 曲线和样条曲线，以及基于统一基础模型的三维模型变形的硬件加速实现方法。

# 第 5 章　运动角色动作驱动技术研究

在三维计算机动画中，把人体作为其中的角色一直是研究者感兴趣的目标，因而关节动画越来越成为人们致力解决的研究课题。近期在这一方面的工作令人惊叹不已，如电影《终结者》和电影《侏罗纪公园》。虽然计算机动画在广告、娱乐、教育、科学计算可视化和仿真等领域占据越来越重要的角色，人体和动物动画的许多问题仍未能很好解决。人体具有 200 个以上的自由度和非常复杂的运动，人的形状不规则，人的肌肉随着人体的运动而变形，人的个性、表情等千变万化。可以说，人体动画是计算机动画中最富挑战性的课题之一。

人们对于虚拟角色的要求不仅需要形象的真实感，也需要运动的真实感，也就是说，虚拟角色的成功仿真取决于两方面的努力，一方面是角色模型变形的效果是否接近于真实，另一方面是角色的动作序列是否协调真实。对于模型变形结果的真实感需要通过使用更优秀的模型和改进变形算法来提高，而角色动作序列的协调性与真实感，需要通过获取更接近于真实的动作数据来保障。

对于模型变形算法，研究者们提出了大量的适用于各种模型的变形方法，在本书的第 3 章和第 4 章中，也对模型变形方法进行了较显著的改进。角色运动的协调性和真实感，需要使虚拟角色拥有与现实世界中相同的运动动作来实现，因此，如何驱动场景中的虚拟角色，使其按照用户希望的方式进行运动，产生效果良好的运动行为是计算机角色仿真的另一重要研究课题。

虚拟角色的运动动作配置一般分为两个层次，即单个动作的真实性和整套动作的连贯性。单个动作的真实性，也称为单个动作的准确性，是指运动过程中每个动作符合实际的骨骼状态，这主要由关节点的位置和骨骼间的夹角来决定。整套动作的连贯性和自然性是指虚拟角色的动作，从视觉角度上接近于真实世界中的动作，除了骨骼状态符合外，还包括该动作发生的时间与现实世界中的动作具有同步性，从而使虚拟角色的运动看起来更为真实。也就是说，在正确的时间里发生正确的动作，这是角色动作设定的最终目的。

目前的动作建立方法有两方面的限制，一方面是这些方法分别使用不同的数据格式，动作捕捉设备获取的动作数据是四元组方式的，视频动作捕捉获取的动作数据是关节点位置方式的，关键帧方式建立的数据是基于旋转角方式的，它们适用的范围各不相同。另一方面是当前的动作数据应用对象，需要在建模过程中，对模型的各段骨骼分别进行建模，而且其建立过程一般是针对特定角色进行，要求数据源的角色拓扑结构和目标角色的拓扑结构和关节点一一对应，从而

在运动仿真过程中对每段骨骼驱动其绕前端关节点旋转，这更进一步限制了运动数据的使用范围。

本章针对虚拟角色的动作配置困难，且现有动作数据的共享性低的问题，提出一种基于动作迁移的角色驱动方法。该方法将专业设备捕捉、视频动作捕捉获得的动作数据，以及现有动画系统中的动作数据，转换为统一的基于关节点旋转角的动作数据。将源角色的动作状态和动作时间一起迁移到指定的虚拟角色上，使两角色在同一时刻具有相同的动作，减少虚拟角色仿真中繁琐的动作配置。通过叠加无配对的关节点旋转角，实现不同拓扑结构的两个角色间可以进行动作迁移。采用“2D→3D→2D”转换方法，配准两角色的姿态，实现从二维角色到二维角色的动作迁移，使动作迁移目标不仅包含三维角色，还可以应用于二维角色。

本章的其他部分内容如下：

（1）介绍关于动作建立和动作驱动的相关技术；

（2）详细给出基于关节点旋转角的动作迁移方法；

（3）介绍基于动作迁移的角色驱动算法具体实现；

（4）通过实验验证通过基于关节点旋转角的动作迁移方法的有效性，用实验数据说明该方法可以使虚拟角色学习现有角色的动作。

## 5.1 相关工作

计算机生成角色仿真的目的是使用虚拟角色配合给定的动作来模拟现实世界中的角色运动。人的视觉系统对运动的物体十分敏感，而对物体的运动之间的细微差别也十分敏感。因此改进虚拟角色的动作数据真实感是运动角色仿真中的重要内容。准确的动作数据，包括动作状态数据、动作时间数据以及两者的准确对应。在虚拟运动角色仿真中，骨骼角色是应用最普遍的一种角色，骨骼结构比较复杂而且相连骨骼之间互相影响，最容易出现不真实的配置结果。一般来说，角色的动作状态较易设定，而动作状态对应的时间却很难设定，一个细微的动作差别都会引起视觉上的巨大反差。

最真实的动作是我们所处的现实世界中的动作，真实的动作数据必须要借鉴或者是来自于现实生活，动作获取常作为角色仿真的动作驱动数据来源。角色动作获取是虚拟现实领域的重要内容，在娱乐、电影和游戏等领域有广泛应用。为更好的模拟人物或动物动作，获取更接近于真实的动作数据尤为重要。传统的运动动作数据建立方法有关键帧技术[8]、专业设备运动捕捉[9]、基于物理的仿真[10]三类方法，基于视频的动作获取方法[27]是新近几年发展起来的低成本的动作数据建立方法。

基于关键帧技术[8]的动作数据建立方法是由操作员设定关键帧中的动作状态和动作发生时间，使用插值算法计算关键帧之间的中间动作状态，实现角色的连续运动。这种方法对于单个动作的骨骼状态，一般可以实现准确的配置，但对于各动作状态的发生时间配置，要想获得满意的仿真效果，需要进行仔细配置与调整，耗费很多时间，而且往往会出现活动动作生硬板滞、明显卡通化的问题，影响虚拟角色的真实感。

为降低动作状态发生时间设定难度，一般通过专用设备捕捉动作数据[9]，在记录动作源角色动作姿态的同时，自动记录该姿态的动作时间，从而获得良好的视觉连续动作序列，并在娱乐、电影和游戏应用中得到广泛应用。目前的大多数运动动作数据也是使用专业设备捕捉获得的[11]。几乎所有的运动捕捉设备，都需要在被获取对象身上放置光学或磁性标记，并使用三角测量的方法取得标记点的空间位置。2005 年 Naksuk[26]使用运动捕捉设备捕捉人体运动数据，并通过关节点间的位置关系将运动迁移到机器人上。设备捕捉到的动作数据一般具有很高的准确性，但专用的捕捉设备受使用环境的制约，大多只能在实验室环境下进行捕捉，在很多场景中无法进行捕捉，不能够广泛应用，而且捕捉过程也受被捕捉对象的运动能力制约。

基于物理的仿真[10]通过正运动学或逆运动学，计算角色运动过程中的动作姿态，不受设备限制也不受目标对象的运动能力限制，通过对关节旋转角设置关键帧，得到相关连各个肢体的位置，这种方法一般称为正向运动学方法。对于一个具有多年经验的专家级动画师，能够用正向运动学方法生成非常逼真的运动。但对于一个普通的动画师来说，通过设置各个关节的关键帧来产生逼真的运动是非常困难的。一种实用的解决方法是通过实时输入设备记录真人各关节的空间运动数据。由于生成的运动基本上是真人运动的复制品，因而效果非常逼真，且能生成许多复杂的运动。

为减少专业设备依赖和弥补动作配置繁琐的缺陷，研究者们提出基于视频的动作捕捉方法[30]，录制现实世界中的人物或动物运动，然后结合所跟踪目标的形状特征和纹理信息，跟踪角色各标定点的位置，从而得到运动数据。该方法有一定的限制，如数据精度不如设备捕捉好，但拥有获取容易、成本低、动作连贯性好、视觉效果真实的特点。2006 年 Li[30]系统地介绍和总结了基于视频的人体运动捕捉研究的技术方法。Brett[122]使用标志点定位捕捉人体动作，并创建动作的骨骼模型，然后基于样本状态实现骨骼体插值变形。2003 年 Yoshimoto[27]使用视频图像提取方法估算特征点的位置，获取人体姿态。Wu[28]基于视频学习方法实现效果十分逼真的鸟类飞行模拟，效果十分逼真。2005 年 Kehl[29]使用多台摄像机捕获人体标定关节点的运动轨迹，建立人体骨骼模型的运动序列。2006 年 Wan[31]使用 3D Graph-cuts 和人体形状模板匹配估算视频中人体姿态。2007 年

Pei[32]基于 ISO-map 的非线性降维和 K-means 聚簇方法，从演讲视频中提取嘴部动作，并通过控制形状向量实现面部变形。Cheung[123]实现了无标识点的视频角色动作迁移。

本章提出一种基于关节点旋转角的动作迁移方法，与以往的动作获取与驱动方法相比，该方法的主要不同之处在于：

（1）将已拥有的动作数据转换为基于关节点旋转角的动作数据，可以很方便地实现两角色的动作状态迁移；

（2）通过旋转角叠加，解决不同拓扑结构的角色之间的动作迁移；

（3）通过 2D→3D→2D 转换，配准两个二维角色的姿态，使动作迁移可以在两个二维角色之间进行。

## 5.2 骨骼角色运动控制技术

指定关节动物的运动，使它能以符合物理规律真实的方式达到给定的目标（如投一个篮球到球筐中）是动画师的目标之一。在计算机动画控制物体运动的方法中，根据指定运动的高低层次，可以分为高层动画方法和低层动画方法，在高层动画中，根据事件及其相互关系来描述物体的运动，如指定一个人从 A 位置移动到 B 位置；而在低层动画方法中，直接指定各个运动参数的值，如直接指定各个关节的旋转角。低层控制通常不考虑运动参数之间的关系，从而使设计运动工作量非常大，通常的解决办法是引入层次运动控制方法，提高描述物体运动的直观性和友好性。

**1. 参数关键帧技术**

关键帧技术是用于运动控制最早的方法，最初仅仅用来插值帧与帧之间卡通画的形状，最后发展成为可以插值任何运动参数。一个好的关键帧插值方法必须能够产生逼真的运动效果并能给用户提供方便有效的控制手段。

**2. 样条驱动动画技术**

样条驱动动画是指先设计好物体的运动轨迹，然后指定物体沿该轨迹运动。物体的运动轨迹为三次样条曲线，并且由用户交互给出。物体的运动可以由速度曲线来控制，速度曲线是弧长对时间的函数，首先从速度曲线上找到给定时间对应的弧长，然后使物体沿轨迹曲线运动该弧长距离。

**3. 物体朝向的欧拉角表示和插值技术**

在物体运动中，关键帧的插值问题实际上可以分为位置插值和朝向插值两个子问题。物体朝向最常见的表示方法为欧拉角，物体的朝向通过绕三个正交坐标轴的旋转来表示，欧拉角按某种特定的次序作用于物体，实际计算时把旋转表示为角位移的形式实现。

**4. 物体朝向的四元数表示和插值技术**

四元数表示矢量和物体的旋转，并且没有冗余信息，它提供了一种比旋转矩阵更为有效的方法。采用四元数插值关键帧朝向的过程可以描述为：

（1）用欧拉角建立关键帧物体的朝向，并转化为单位四元数；

（2）根据关键帧四元数，生成插值四元数曲线；

（3）计算 $t$ 时刻的四元数，并将其转化为旋转矩阵。

**5. 关节动画驱动技术**

关节动画是计算机动画中最具挑战性的课题之一，它的主要目的是模拟骨骼动物（尤其是人）的运动。与普通的三维动画技术相比，该技术涉及的建模、运动控制和绘制三个过程均较为复杂，因此成为目前计算机动画研究中最为活跃的领域之一。

关节动画中的关节链是运动控制的主要结构，关节链是由一系列依次相连的物体组成的，两物体之间的连接点称为关节，两个相邻关节之间的物体称为链杆，可以抽象为一条直线段。关节链的运动控制较常用的方法主要有 DH 表示法和 AP 表示法。DH 表示法通过对每一个链杆建立坐标标架来描述链杆相对于其相邻链杆的运动，四个独立参数定义相邻链杆之间的线性变化关系，它们分别是链杆的长度、相邻链杆的距离、扭角和夹角。AP 表示法存储如下信息：关节的位置、关节轴线方向、指向关节所连的链杆的指针，共七个参数，其中包括三个位置参量、三个关节轴线方向和一个关节角度参量。

## 5.3　基于关节点旋转角的动作迁移方法

为将获取到的动作数据应用到目标角色上，可以使用动作迁移将动作状态从一个模型迁移到另一个模型上，从而共享或共用动作序列，减少动作数据对模型的依赖性，并避免重复动作获取和状态配置。动作迁移还可以实现针对无法进行动作获取的角色动作数据生成，如计算机动画和电影制作领域中的怪物等角色。

骨骼类动物的各种动作，一般通过各段骨骼围绕关节点旋转来完成，而骨骼动物的动作状态，也主要是由骨骼之间的关节点旋转角决定的。每个关节点由上端骨骼的方向顺时针旋转到下端的骨骼方向的旋转角度，称之为关节点的旋转角。关节点旋转角表示该关节点前后的渲染方向的变化，也表征该关节点前后局部坐标系与全局坐标系的转换关系的变化。通过当前关节点以及两侧关节点的位置，可以求得当前关节点的旋转角，反之，通过关节点旋转角和前两个关节点的位置亦可求得下端关节点的位置，通过基于关节点旋转角的迁移策略，可以实现动作数据在不同模型之间的动作迁移，从而实现使用源角色的动作数据驱动目标虚拟角色。

### 5.3.1 基于关节点旋转角的动作迁移原理

动作迁移过程中的关节点对应关系如图 5-1 所示，图中源角色的关节点 $P_0P_1P_2P_3$，分别对应至目标角色的关节点 $P'_0P'_1P'_2P'_3$。以 $P_2$ 关节点为例，在动作迁移过程中，通过 $P_1P_2P_3$ 位置计算 $P_2$ 的旋转角，将该旋转角应用到 $P'_2$关节点上，通过 $P'_1P'_2$骨骼段的 3D 长度，计算关节点 $P'_3$的位置，由此可以实现两个角色之间的动作同步。

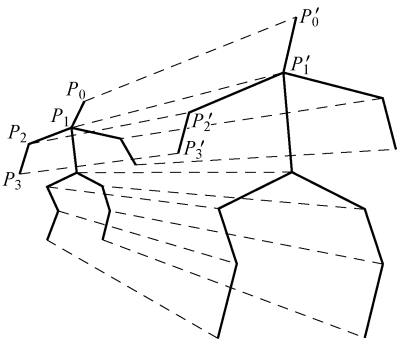



图 5-1 动作迁移中的关节点对应关系示意图

不同拓扑结构角色的动作迁移原理如图 5-2 所示，在图中的 $P_0P_1P_2P_3$ 是源角色中的关节点，分别对应目标角色中的 $Q_0Q_2Q_3Q_4$ 四个关节点，而 $Q_1Q_5$ 两个关节点没有源角色中的关节点进行对应。在这种情况下进行动作迁移时，通过保持 $Q_1$ 和 $Q_5$ 的旋转角不变，而将 $Q_1$ 和 $Q_5$ 的旋转角分别叠加到 $Q_0$ 和 $Q_4$ 的旋转角上，使 $Q_0Q_1Q_2$ 三个关节点组成一个整体，类似于一条骨骼 $Q_0Q_2$，而 $Q_5$ 作为 $Q_3Q_4$ 的延伸部分，随 $Q_3Q_4$ 的动作进行运动。反之，当源角色中存在无对应的关节点时，同样将该关节点的旋转角叠加到前一关节点上，实现拓扑结构的同化。

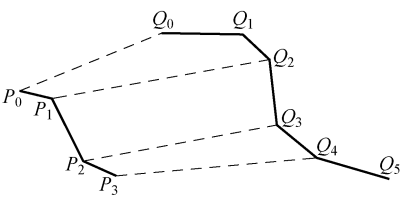



图 5-2 不同拓扑结构角色的动作迁移示意图

动作迁移主要是将角色动作从源角色变换到目标角色上，使两角色拥有相同的动作状态，该操作可以在拓扑相近或略有不同的骨骼框架之间进行，迁移的源角色和目标角色的关节点基本对应即可。以图 5-2 中所表示的动作迁移过程为例，其动作状态同步过程如图 5-3 所示，首先根据 $P_0P_1P_2$ 求得 $P_1$ 关节点的旋转角，

然后根据此旋转角、$Q_0Q_2$ 的位置和 $Q_2Q_3$ 的长度，即可求得 $Q_3$ 的位置。

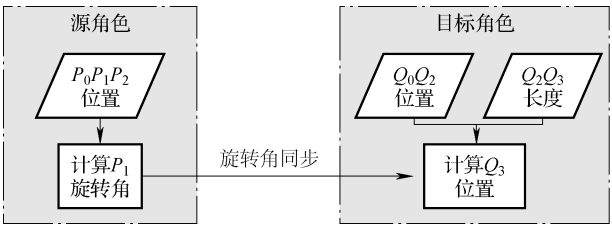



图 5-3　动作迁移的旋转角同步示意图

## 5.3.2　二维动作状态的三维重建

通过三维专业动作捕捉设备可以直接获得三维动作数据，而通过视频跟踪或者二维动画获取的动作数据，它们的一个特点是只有二维平面信息而没有三维深度信息。二维动作数据包括源角色的关键关节点的平面位置，要将这些动作状态应用到三维角色的仿真中，必须要进行二维到三维的信息转换，也即三维重建。

根据三维空间中角色骨骼长度不变的性质，由骨骼的二维投影长度计算各骨骼的前后向倾斜角，然后根据骨骼长度计算关节点的深度信息，最终得到对应该二维动作数据的三维动作数据。动作状态的三维重建，根据动作状态的二维数据求解关节点的深度信息，动作状态的三维重建原理如图 5-4 所示。

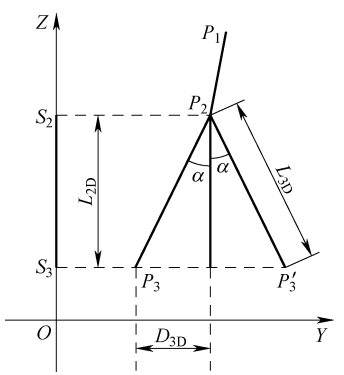



图 5-4　二维动作状态的三维重建示意图

以图 5-4 中所示的骨骼二维投影 $S_2S_3$ 为例，在投影平面上的二维投影长度为 $L_{2D}$，在角色动作过程中，该投影对应的骨骼三维长度 $L_{3D}$ 不变，$L_{2D}$ 与 $L_{3D}$ 的长度关系会随该骨骼与平面的夹角 $\alpha$ 有关，关节点 $P_3$ 相对于上一关节点 $P_2$ 的深度差 $D_{3D}$ 可用式（5-1）求得。

$$\begin{aligned} D_{3D} &= L_{3D}\sin\alpha \\ \alpha &= \arccos(L_{2D}/L_{3D}) \end{aligned} \tag{5-1}$$

根据此深度信息可以获得该关节点的两个可能位置，如图 5-4 中的 $P_3$ 和 $P_3'$，具体应用时需要根据合理性选择其中一个。

## 5.3.3　以二维角色为目标的动作迁移

在计算机虚拟角色仿真中，二维与三维角色模型都能进行较真实的动作模

拟，但其也有许多不同点。三维角色模型具有变形结果精度高、细节明显、表现力强等特点，但三维模型建模过程与变形计算复杂、控制繁琐，由于渲染效率不如二维图像高，因此变形结果渲染的运行系统要求较高。图像形式的二维虚拟角色，具有获取方便、变形计算量小、渲染对设备需求低的特点，尤其重要的是二维虚拟角色拥有庞大的现有资源库，而且制作门槛低可以快速制作。由于以上原因，虽然三维角色具有二维角色无法比拟的表现力，但二维角色仿真仍然具有极大的生命力。

本书研究的内容是大规模运动角色仿真，在该仿真应用中，最关注的是整体场景效果以及渲染的实时性，当角色数量较多时，三维角色模型渲染的帧速率过低，无法实现实时仿真，而二维虚拟角色在该应用方面具有很大优势。在本章的动作迁移过程中，二维虚拟角色与三维虚拟角色将同时作为迁移目标对象，应用于大规模运动角色的快速仿真。

相比于将动作迁移到三维角色模型，以二维虚拟角色为目标的动作迁移遇到的新困难及本章的动作迁移方法的解决方案为：

（1）二维虚拟角色只有平面信息，单一图像不能实现肢体旋转等大角度旋转动作，本章的方法通过切换二维虚拟角色的图像实现该类动作；

（2）对于从二维动画中或视频中获得的二维运动数据，由于无法保证源角色与虚拟角色的姿态一致，如一个偏左一个偏右，使得两者的动作迁移成为困难。本章的动作迁移通过“2D→3D→2D”转换，在三维角色空间中进行姿态配准，实现两者关节点旋转角的传递。

### 5.3.4 源角色与目标角色的关节点对应

通过基于关节点旋转角的同步策略，将源角色的动作状态迁移到拓扑结构相近的虚拟角色模型上，然后使虚拟角色与源角色在同一时间具有相同的动作，这样就实现了两角色的动作时间和动作状态同步迁移，从而驱动虚拟角色自动运动，避免角色仿真中繁琐的动作配置。

动作迁移的过程需要两方面的数据，一方面是源角色的运动数据，另一方面是源角色与目标角色的关节点对应关系。图 5-5 为基于视频的动作迁移方法中视频角色和 2D 目标角色的骨骼关节点对应关系图。在动作迁移的初始化过程中，需要指定两角色的关节点对应关系。

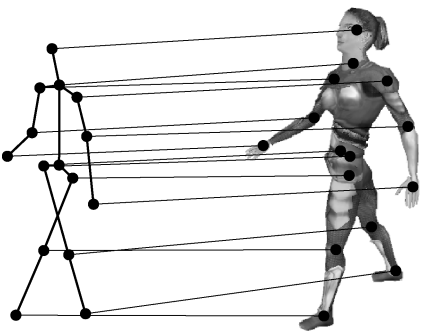


图 5-5 动作状态与目标角色的关节点对应关系图

### 5.3.5　关键动作状态迁移

从源角色的运动中获得的骨骼状态称为关键动作状态，是角色运动过程中标准的动作状态，类似于动画制作中的关键帧。关键动作状态的迁移过程可描述为

（1）针对源角色的每个关节点，根据关节点位置和前后两侧关节点位置，计算该关节点的旋转角；

（2）将旋转角同步到目标角色的对应关节点上；

（3）旋转角同步完毕，针对每一条关节链，根据旋转角更新关节点的位置，从而求得整个骨骼框架中的关节点新位置。

关键动作状态迁移中目标角色关节点位置求取的原理如图 5-6 所示，通过同步旋转角的方法，剔除了两角色骨骼长度不同的影响。图中的 $P_0P_1P_2P_3$ 为源角色的一条关节链，$Q_0Q_1Q_2Q_3$ 为目标角色中与之对应的关节链。

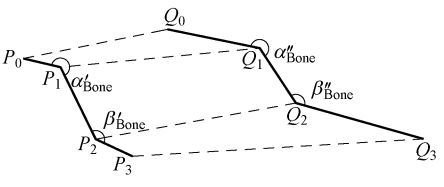



图 5-6　关键动作状态迁移原理图

根据其骨骼框架查找各关节点两侧相连的骨骼，由相连的三个关节点的位置，计算中间关节点的旋转角，该旋转角用式（5-2）进行计算，即

$$\alpha_{\text{Bone}} = \begin{cases} \alpha_{\text{Bet}} & ,flag \geqslant 0 \\ 2\pi - \alpha_{\text{Bet}} & ,flag < 0 \end{cases}$$
$$\alpha_{\text{Bet}} = \arccos\left(\frac{\overrightarrow{p_0p_1} \cdot \overrightarrow{p_0p_r}}{length_1 \cdot length_r}\right) \tag{5-2}$$

式中，$\alpha_{\text{Bet}}$是两骨骼之间的几何夹角；$\overrightarrow{p_0p_1}$和$\overrightarrow{p_0p_r}$分别是从当前关节点 $P_0$ 出发的两侧骨骼向量；$length_1$和 $length_r$分别是它们的长度；$flag$ 是旋转角是否大于 180°的标志，其计算见式（5-3），$(x_0, y_0)$、$(x_1, y_1)$和$(x_r, y_r)$分别是当前关节点与上下关节点在视频中的对应点位置坐标。

$$flag = (y_1 - y_0)(x_r - x_0) - (x_1 - x_0)(y_r - y_0) \tag{5-3}$$

根据旋转角 $\alpha_{\text{Bone}}$和虚拟角色的骨骼长度，可以利用式（5-4）计算当前关节点的下一关节点位置，其中 $\alpha$ 即为关节点旋转角。

$$P_r = P_0 + \frac{length_r}{length_1}\left(\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix} \times \overrightarrow{p_0p_1}\right) \tag{5-4}$$

当某一关节点的旋转角发生变化，则从该点开始，同一链的关节点都需要重新计算其位置，最终可以得到对应于当前动作状态的虚拟角色动作状态。

关键动作状态的迁移步骤可以描述为

（1）通过 $P_0P_1P_2$ 点的位置计算 $\alpha'_{\text{Bone}}$；

（2）使用该角度设定 2D 虚拟角色中 $Q_1$ 的旋转角，使 $\alpha''_{\text{Bone}} = \alpha'_{\text{Bone}}$；

（3）根据 $Q_0Q_1$ 的位置计算 $Q_2$ 的新位置；

（4）使 $\beta''_{\text{Bone}} = \beta'_{\text{Bone}}$，实现 $Q_0Q_1Q_2$ 部分的姿态与 $P_1P_2P_3$ 部分相同。

### 5.3.6 任意时刻的动作状态计算

通过关键动作状态的迁移，将源角色动作状态迁移至目标角色，每个关键动作状态包括该时刻的关节点位置及其旋转角。在目标角色运动过程中，为使虚拟角色可以连续动作，需要计算任意时刻的动作数据。本方法在关键状态之间使用关节点旋转角插值的方法获得任意时刻的旋转角，从而获得任意时刻的动作状态驱动数据。

角色运动中的任一时刻必须定位于两个关键状态时间之间，该时刻动作状态的计算过程是，首先确定该时刻两侧的关键状态；然后根据时间比例线性插值关节点的旋转角，获得该时刻的动作状态。以某关节点为例，所求状态的运行时间为 $t$，$t_{\text{A}}$ 和 $t_{\text{B}}$ 分别为该时间两侧关键状态的运行时间，$\alpha_{\text{A}}$ 和 $\alpha_{\text{B}}$ 分别为两状态中该关节点的旋转角，中间动作状态的关节点旋转角 $\alpha$ 可以用式（5-5）计算，即

$$\begin{aligned} \alpha &= (1-\tau)\alpha_{\text{A}} + \tau\alpha_{\text{B}} \\ \tau &= (t - t_{\text{A}})/(t_{\text{B}} - t_{\text{A}}) \end{aligned} \tag{5-5}$$

## 5.4 基于动作迁移的角色驱动算法实现

基于本章的思想和方法，结合本书第 3 章和第 4 章的角色变形方法，现已实现了基于视频迁移的角色驱动算法，算法的执行过程包括角色驱动与变形初始化和角色运动仿真两部分，具体的算法实现可以描述如下：

（1）角色驱动与变形初始化　加载目标虚拟角色，在目标角色中标定控制关节点，连接关节点形成变形区域，建立简化骨骼模型。加载源角色及其运动数据，对应源角色和目标虚拟角色的关节点。

（2）视频动作仿真　首先，进行动作状态迁移。在每帧视频动作迁移中，加载源角色动作数据，根据关节点位置计算各关节点的旋转角；将该旋转角传递给目标虚拟角色的对应关节点旋转角。然后，按关节点的连接次序，重新计算目标角色所有关节点的位置，获得对应该源角色动作状态的目标角色状态，关键动作状态迁移完毕。最后，进行角色仿真与绘制。动作状态迁移完毕之后，以虚拟角色各关节点的位置作为输入，实现目标角色的动作仿真。图 5-7 所示为角色动作驱动的数据流图。

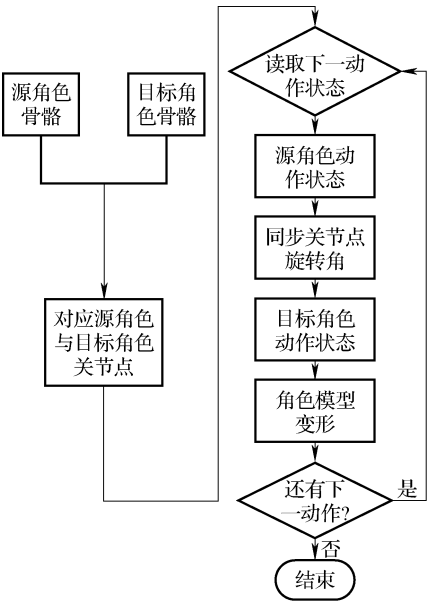



图 5-7　角色动作驱动的数据流图

## 5.5　动作驱动关键代码

本章的动作驱动技术，可以完成在不同模型上进行动作迁移，只需要相互迁移的两个模型大体类似即可。两模型之间的模型差异通过关节点合并互相抵消，然后骨骼长度之间的差异通过基于关节点旋转角的方法完成动作迁移。

本章中实现的动作驱动关键代码有如下几个：动作关节的旋转角计算、关节点旋转角的迁移（即目标模型关节点的位置计算）、中间帧驱动信息的计算以及自动驱动算法。

### 5.5.1　目标模型关节点位置计算

当将源模型的关节点旋转角被迁移到目标模型上之后，需要根据获得的旋转角计算每一条骨骼链上所有关节点的新位置。计算过程首先根据设置好的旋转角计算当前关节点的其他数据，并对同链中所有关节点更新其信息。根据旋转角更新所有关节点位置的代码如下：

```
bool reCalJptByAngleDir()
{
        //根据 jptSetFlg 数组，设置点按 angleDir 计算
```

```
//未设置点按 angleBet 计算
float angleSum;
float dx,dy;
SkeJptDetail * provjpt,* curjpt,* nextjpt;
SkeChain * pca = chainArray;
bool isDirOrBet,isDirStart;
int p2vx,p2vy;
int * ptindex;
for(int i = 0;i < chainNum; + + i)
{
    isDirStart = false;
    ptindex = pca -> ptIndex;
    //每条链从第三个点开始计算，前两个点肯定不会移动的
    //至 nextjpt 移动到最后一个止，最后一点，已不需要重新计算
    for(int j = 1;j < pca -> ptNum - 1; + + j)
    {
        isDirOrBet = jptSetFlg[ptindex[j]];
        if(isDirOrBet) isDirStart = true;
        if(! isDirStart) continue;//仍未开始则跳过
        provjpt = jptArray + ptindex[j - 1];
        curjpt = jptArray + ptindex[j];
        nextjpt = jptArray + ptindex[j + 1];
        if(isDirOrBet)
        {
            //如果有方向就按方向
            angleSum = curjpt -> angleDir3;
            dx = curjpt -> length3* cosf(angleSum* 3.14159265f/180.0f);
            dy = curjpt -> length3* sinf(angleSum* 3.14159265f/180.0f);
            nextjpt -> x = curjpt -> x + dx;
            nextjpt -> y = curjpt -> y + dy;
            nextjpt -> angleDir2 = angleSum - 180;
            //计算 alphaBet,rotAngle1,rotAngle2,isLess180
            p2vx = provjpt -> x - curjpt -> x;
            p2vy = provjpt -> y - curjpt -> y;
            curjpt -> isLess180 = (p2vy* dx - dy* p2vx) > = 0;
```

```
                curjpt -> alphaBet = acosf((float)(p2vx * dx +
                        p2vy * dy)/sqrtf((float)(p2vx * p2vx +
                        p2vy * p2vy) * (dx * dx + dy * dy))) *
                        180.0f/3.14159265f;
                //这里借用是否超过 180°的条件
               curjpt -> rotAngle1 = (curjpt -> isLess180? (180 -
                     curjpt -> alphaBet/2): -(180 - curjpt ->
                     alphaBet/2));
                curjpt -> rotAngle2 = (curjpt -> isLess180?
                        -curjpt ->alphaBet/2: curjpt ->alphaBet/2);
                }
                else
                {
                    //如果不是骨骼方向就使用骨骼夹角进行计算
                    //每链的第一个需计算的 curjpt 肯定是 Dir
                    if (curjpt -> isLess180)
                        angleSum += 180 - curjpt -> alphaBet;
                    else
                        angleSum -= 180 - curjpt -> alphaBet;
                    dx = curjpt ->length3 * cosf(angleSum * 3.14159265f/180.0f);
                    dy = curjpt ->length3 * sinf(angleSum * 3.14159265f/180.0f);
                    nextjpt -> x = curjpt -> x + dx;
                    nextjpt -> y = curjpt -> y + dy;
                    nextjpt -> angleDir2 = angleSum - 180;
                    //计算 angleDir3
                    curjpt -> angleDir3 = angleSum;
                } //if isDirorBet
            } //for j
            ++pca;
    } //for i
    return true;
}
```

## 5.5.2　中间帧计算算法

在动作迁移过程中，目标模型通过动作迁移算法从源模型获得的动作数据一

定是标准的，这些动作帧称为关键帧。相邻关键帧之间有一定的时间差，为实现目标模型的运动平滑性，需要将相邻关键帧之间的中间帧数据计算出来。中间帧的计算首先判定当前时间，处于哪两个时间节点中间，并计算该中间帧与两侧关键帧的时间差；然后根据前后两个状态，计算当前状态，计算方法可以自由选择。本章所采用的中间帧计算代码如下：

```
bool calCurDriveStatus()
 {
    char tstr[MAXPATH];
    //当前时间 curDrive.time
    SkeDrive * pleft = skedrive + driveNum - 1;
    //时间循环，curTime 在 pleft 和 pright 两个状态之间
    float curTime = curDrive.time;
    while(curTime > = pleft -> time)
    curTime - = pleft -> time;
    pleft = pright - 1;
    SkeJptDetail * pjp;
    if((curTime > = pright -> time) ||(curTime < pleft -> time))
    {
        pright = skedrive;
        for(int i = 0; i < driveNum; + + i)
        {
          if(pright -> time > curTime)break;
           + + pright;
        }
        pleft = pright - 1;
        //改变过间隔，则更新 jptSetFlg 标记，和更新旧点方向角
        //更新旧点方向角，为左侧值
        for(int i = 0;i < jptNum; + + i)
        {
            pjp = jptArr + i;
            leftADir[i] = pjp -> angleDir3;
        }
        //更新 jptSetFlg 为右侧值，用于传递给 Ske 类
        //更新 curDirve 的结构
        curDrive.ptNum = pright -> ptNum;
```

```
            memcpy_s (curDrive.ptIndex, pright ->ptNum*  (sizeof (int)),
                    pright ->ptIndex,pright ->ptNum* (sizeof(int)));
            //调整 leftADir，使其各方向与 pright 的方向差小于 360°
            int inx; float diff;
            for(int i =0; i <pright ->ptNum; + +i)
            {
                inx =pright ->ptIndex [i];
                diff =leftADir[inx] -pright ->angleDir[i];
                while(fabsf(diff) >180){
                    if(diff >180)
                        leftADir[inx] - =360;
                    else if(diff  < = -180)
                        leftADir[inx] + =360;
                    diff =leftADir[ inx ] -pright ->angleDir[i];
                }
            }
        }
        //建立一个夹角的数组，用夹角，那就得根据方向角计算夹角
        //当前时间离左侧状态的比例
        float leftRatio =(curTime -pleft ->time)/(pright ->time -
                pleft ->time);
        for(int i =0; i <pright ->ptNum; + +i)
        {
              curDrive.angleDir[ i ] = (1 - leftRatio) *  leftADir
              [pright - > ptIndex[ i ]] + leftRatio *  pright - > an-
              gleDir[i];
        }
        return true;
    }
```

### 5.5.3　顶点数据计算

目标模型的关节点位置，根据所迁移过来的动作驱动数据更新之后，需要根据新的关节点信息计算目标模型控制网格，以实现最终的模型变形。根据关节点信息计算目标模型控制的代码如下，为进一步使用 GPGPU 进行加速实现，本章列出其 CPU 模拟代码。

```
bool calVertData(float * O2ptXY,float * I4lwDiviN,float ptDivNum,
        float shape)
{
    //输入参数：Alpha，if (isLess180) 则>0，else <0;
    //ptDivNum，shape：uniform，还有一个是modelviewproj
    //首先提取计算所需变量
    int ptInx = (int)(I4lwDiviN[3]);
    float lenC = I4lwDiviN[0],widC = I4lwDiviN[1];
    float ptnum = I4lwDiviN[2];
    float * I4ptXYDirAlpha = pfptXYDirAlpha4 + 4* ptInx;
    float alpha1 = I4ptXYDirAlpha[3]/2* 3.14159265/180;
    float ptDir = (I4ptXYDirAlpha[2] + I4ptXYDirAlpha[3]/2)*
            3.14159265/180;
    if(ptnum>0 && alpha1>0) widC = -widC;
    //然后求Yend
    float sinV = sinf(alpha1),cosV = cosf(alpha1);
    float signW = alpha1*  (lenC +1);
    float yend = -lenC* sinV + fabsf(signW)/signW* widC* cosV;
    float xCur, yCur;
    if (ptnum<0)
    {
        //矩形部分
        yCur = yend;
        xCur = fabsf(lenC)* cosV - widC* sinV;
    }
    else if (yend* lenC* alpha1>0) {
        yCur = 0;
        xCur = fabsf(lenC)* cosV - widC* sinV;
    }
    else
    {
        //曲线部分
        float aC = fabsf (widC* shape/sinV);
        float bC = fabsf (widC* shape/cosV);
        float iX = widC/fabsf (sinV);
```

```
        float tX = aC * sqrtf(lenC* lenC* sinV* sinV/bC/bC +1) -
                fabsf(lenC* cosV);
        yCur = yend* ptnum/ptDivNum;
        xCur = aC* sqrtf (yCur* yCur/bC/bC +1) - tX + iX;
    }
    sinV = sinf (ptDir); cosV = cosf (ptDir);
    if (I4ptXYDirAlpha [0] ==597) yend =1;
    O2ptXY[0] = xCur* cosV - yCur* sinV + I4ptXYDirAlpha[0];
    O2ptXY[1] = xCur* sinV + yCur* cosV + I4ptXYDirAlpha[1];
    return true;
}
```

### 5.5.4　角色运动自动驱动算法

本章中的动作迁移算法，是根据加载的源模型的自动变形信息，实现目标模型的自动驱动变形。本章算法中所使用的总调算法的代码如下：

```
void driveAutoMotion()
{
    ASSERT(motiondata! =NULL);
    //步进变形的时间，并设置到 Data 类中
    static int curCount =0;
    curTime = curCount* 0.01;
    if(curTime > maxTime){
        curTime = maxTime;
        curCount =0;
    }else{
        + +curCount;
    }
    motiondata -> curDrive.time = curTime;
    //调用 Data 类，计算当前时间的驱动状态
    motiondata -> calCurDriveStatus ();
    //根据驱动信息对 Ske 类设置骨骼
    //将 motiondata 类中的 drive 信息，送到 skedata 类中
    SkeDrive * pdrv = &(motiondata -> curDrive);
    skedata - >setAngleDir(pdrv -> ptNum,pdrv -> ptIndex,
            pdrv -> angleDir);
```

```
        //调用 skedata 计算各关节点的新位置
        skedata -> reCalJptByAngleDir();
        //最后就是渲染变形后的结果
        //调用变形后网格计算
        motiondata -> needReArrange = true;
        motiondata -> dataReArrange();
        calmesh -> cCalMeshData2(1);
        return;
    }
```

## 5.6 实验结果与分析

基于本章的角色驱动的思想与方法，现已经在 PC 上实现了一个基于动作迁移的角色驱动演示程序。本章实验的硬件配置为 Intel P4 3.0G CPU，1GB 内存，nVidia GeForce FX6600 128M 显卡，软件环境为 Windows XP 操作系统，编程环境为 Microsoft Visual Studio 2005，3D 开发环境为 OpenGL。

本节的实验，主要包含如下几部分：

（1）验证基于动作迁移的角色驱动方法的可行性。本章提出的基于动作迁移的角色驱动方法，采用设备捕捉、视频跟踪及关键帧动画等作为动作数据源，通过移植源角色的动作避免繁琐的动作配置，并完成与源角色相同的动作序列。通过对比实验中的源角色姿态和目标虚拟角色姿态，说明本章动作驱动方法的可行性。

（2）获得在基于视频的动作驱动方法中各步骤的运行时间。本章提出的角色动作驱动方法，通过同步各关节点的旋转角，实现两角色间的动作迁移，通过连续多次实验，获取各部分的执行时间。

### 5.6.1 基于动作迁移的角色驱动结果

本章提出的基于动作迁移的角色驱动方法，将源角色动作状态和动作时间一起迁移到指定的虚拟角色上，因此，目标虚拟角色与源角色在同一时刻应该具有相同的动作，本章动作驱动方法的可行性，可以通过对比实验中的源视频角色姿态和目标虚拟角色姿态来说明。

本节使用两组实验分别验证从视频运动数据和关键帧运动数据作为数据源的角色驱动可行性，第一组实验使用一段行人走路的视频运动数据作为数据源，目标角色为 Cally 二维角色。第二组实验使用一段舞蹈的关键帧运动数据作为数据源，目标角色为 Girl 二维角色。

**实验 1：从骨骼动作状态到 Cally 角色的动作驱动**

图 5-8 所示为动作序列的骨骼状态。每一帧状态对应的动作数据包括运动时间和骨骼状态两部分，该数据将作为动作迁移过程的输入数据。图 5-9 所示为通过基于关节点旋转角的动作迁移方法实现的 Cally 运动仿真效果，其中角色变形部分使用本书第 3 章中的基于自适应网格的二维角色变形方法完成，通过动作数据图 5-8 和运行结果图 5-9 的对比，可以看出，两者之间的动作能够实现完全同步。

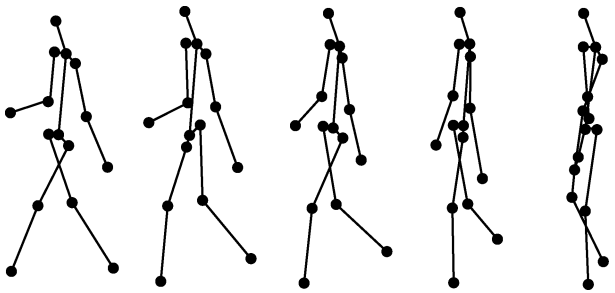


图 5-8　动作序列的骨骼状态

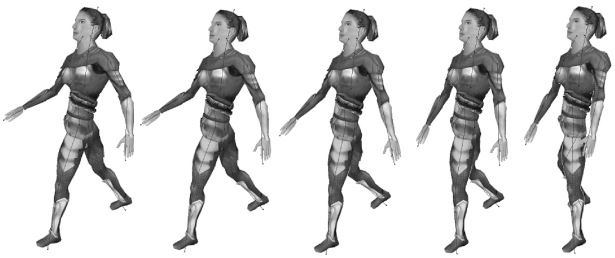


图 5-9　Cally 角色的运动仿真效果

**实验 2：从运动数据到 Girl 角色的动作迁移**

第二组实验是从其他应用中转换的运动数据驱动目标角色的运动，预先转换现有的动作数据为关节点旋转角，所得的运动数据见表 5-1。

**表 5-1　角色初始状态的运动数据**

| 关节点序号 | 关节点旋转角/° | 关节点序号 | 关节点旋转角/° |
|---|---|---|---|
| 0 | 209. 7 | 8 | 119. 3 |
| 1 | 182. 9 | 9 | 149. 3 |
| 2 | 188. 3 | 10 | 179. 5 |
| 3 | 0 | 11 | 0 |
| 4 | 23. 86 | 12 | 92. 05 |
| 5 | 185. 8 | 13 | 160. 1 |
| 6 | 173. 4 | 14 | 174. 3 |
| 7 | 0 | 15 | 0 |

在基于关节点旋转角的角色运动数据中，包含了每一个运动时刻更新的关节点序号和关节点新旋转角，在表 5-2 中给出了四个时刻的运动数据。

**表 5-2 基于关节点旋转角的角色运动数据**

| 动作时间/s | 关节点个数 | 关节点 1（旋转角/°） | 关节点 2（旋转角/°） | 关节点 3（旋转角/°） | 关节点 4（旋转角/°） | 关节点 5（旋转角/°） |
|---|---|---|---|---|---|---|
| 0.1 | 4 | 1（176.1） | 2（150.5） | 5（1.023） | 6（327.9） | / |
| 0.2 | 4 | 1（153.1） | 2（111.3） | 5（336.2） | 6（286.2） | / |
| 0.3 | 4 | 1（182.4） | 2（186.6） | 5（11.65） | 6（5.595） | / |
| 0.4 | 4 | 1（214.6） | 2（259.3） | 5（50.63） | 6（92.26） | / |
| 0.5 | 4 | 1（172.4） | 2（165.9） | 5（5.194） | 6（359.7） | / |
| 0.6 | 5 | 1（149.3） | 5（333.1） | 6（304.4） | 13（135.6） | 14（163.5） |
| 0.7 | 5 | 1（200.6） | 2（227.6） | 5（13.72） | 6（25.06） | 13（73.84） |
| 0.8 | 5 | 1（226.3） | 5（42.56） | 6（66.50） | 13（49.40） | 14（4.609） |

通过基于关节点旋转角的动作驱动方法实现的 Girl 运动仿真效果如图 5-10 所示，角色变形部分使用本书第 3 章中的基于自适应网格的二维角色变形方法完成。

图 5-10 Girl 角色的运动仿真效果

## 5.6.2 动作驱动执行时间的分析

本章的实验针对各步骤的运行时间进行分析，对 Cally 模型进行动作驱动实验，每次实验连续执行 100 帧，各关键计算步骤执行时间及总时间记录见表 5-3。实验数据表明，对于 20 个左右关节点的普通模型，每帧计算中每个模型的中间状态计算、虚拟角色变形、虚拟角色渲染三部分的时间均在 1.2ms 左右，关节点位置更新计算的时间在 0.1ms 左右，计算总时间保持在 3～4ms，可以应用于大量动作角色的场景仿真，实现实时动作迁移和渲染。

表 5-3　动作驱动过程各步骤执行时间　　（单位：ms）

| 骨骼状态计算 | 目标角色变形 | 目标角色绘制 | 执行总时间 |
|---|---|---|---|
| 1. 12681 | 1. 149 | 1. 201 | 3. 552 |
| 0. 96533 | 1. 137 | 1. 187 | 3. 346 |
| 1. 14478 | 1. 168 | 1. 218 | 3. 607 |
| 1. 17389 | 1. 161 | 1. 197 | 3. 590 |
| 0. 98755 | 1. 125 | 1. 179 | 3. 346 |
| 1. 12877 | 1. 172 | 1. 205 | 3. 580 |
| 1. 17085 | 1. 156 | 1. 199 | 3. 586 |
| 1. 15962 | 1. 183 | 1. 225 | 3. 659 |
| 1. 08404 | 1. 156 | 1. 216 | 3. 525 |
| 1. 08874 | 1. 150 | 1. 197 | 3. 501 |
| 1. 09134 | 1. 177 | 1. 217 | 3. 550 |
| 1. 10317 | 1. 155 | 1. 212 | 3. 532 |

根据表 5-3 中的数据所绘制出的动作迁移的各关键步骤执行时间曲线如图 5-11 所示。

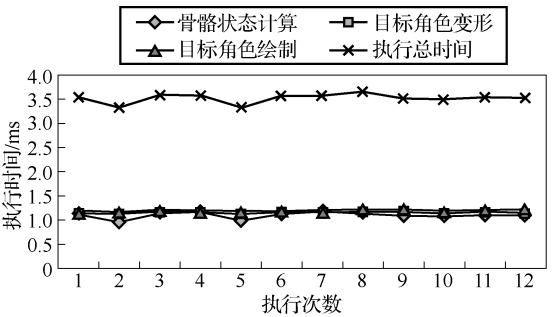



图 5-11　动作驱动关键步骤计算时间

## 5.7　本章小结

本章提出了一种基于动作迁移的角色驱动方法，解决了运动角色仿真中虚拟角色的动作配置困难、动作数据共享性差的问题。该方法从设备捕捉、视频跟踪、关键帧动画等数据源中获得动作数据，将这些转换为统一的基于关节点旋转角的动作格式；使用基于旋转角叠加的方式，实现在源角色和目标角色的骨骼拓扑结构不一致情况下的动作迁移；通过关节点旋转角传递，将源角色动作迁移到

虚拟角色上，使两角色在同一时间动作相同，减少角色仿真中繁琐的动作配置；通过二维/三维的数据转换配准角色姿态，实现两个二维角色之间的动作迁移。

实验证明，该方法可以使用虚拟角色有效仿真源角色的运动，解决以往虚拟角色仿真中的动作配置困难等问题。

作者进一步的工作将探讨角色动作状态的空间转换方法，并研究复杂动作序列的二维角色仿真方法。

# 第 6 章　角色群组快速绘制技术研究

随着人们欣赏水平的逐步提高，特效技术的应用越来越普遍，并已经成为一种必需的手段。近年来电影的发展正在走向大场景、多角色等宏大场面，如电影《指环王 3：王者无敌》中 20 万人的战争场面，就是一个大规模运动角色仿真的典型应用。

大规模运动角色仿真在团体操、大型操等大型演练项目中也经常使用，在这些项目中，每个人有预设的路线和动作，几千人共同达到一个整体效果。这样的应用使每一次排演的准备时间和排练时间都很长，对于导演的想象力发挥造成了很大的限制。借助于群组仿真技术，这种大型演练项目可以进行事先编排，提前发现问题，并修改其中的不足，达到最佳效果，如 2008 年北京奥运会开幕式中，许多个场景的编排都借助了群组仿真技术。

大规模运动角色仿真需要在计算机中以三维的方式逼真地绘制运动角色的仿真结果。一个普通精细度的三维人体模型大约包含 5000 个面片，而场景中往往会有几千个甚至上万个角色在同时运动，角色绘制的速度对仿真速度的影响举足轻重。在现有的硬件条件下，利用常规的方法实时绘制成百上千甚至上万个三维模型几乎不可能，大规模运动角色场景的快速高效仿真方法亟需研究解决。

对于绘制能力的提高，一般有 LOD 层次细节方法和基于图像的渲染两种方法。LOD 层次细节技术通过减少需绘制的多边形数量，提高模型绘制的效率，可以实现 100 约个角色的实时绘制，但当场景中有上千个角色时，仍然无法实现实时计算和绘制。基于图像的渲染能够快速绘制场景，但只能绘制静态的景物，很大程度上限制了它的应用范围。

本章针对大规模运动角色的绘制问题，提出了一种结合 Billboard 技术和动态纹理技术的快速绘制方法。本方法通过在场景中复用较少的实际角色降低运动角色的规模。通过渲染到纹理技术将角色运动过程实时渲染到一个动态纹理中，并将该纹理应用到 Billboard 上，从而快速渲染场景中的运动角色。在运动角色绘制过程中，使用 GPGPU 技术加速 Billboard 顶点位置计算。实验证明，该方法可以高效地仿真大规模运动角色场景，可以实现包含 5000 个角色的场景实时绘制。本章提出的快速绘制技术，结合本书提出的二维/三维角色变形方法和角色驱动方法，可以实现大规模运动角色的实时仿真。

本章的其他部分将包含如下内容：

（1）分析目前大规模角色仿真的技术现状，以及当前常用的加速绘制方法，

详细介绍基于 Billboard 的加速绘制方法；

（2）详细介绍基于动态纹理的快速绘制技术，包括动态纹理生成、基于动态纹理的绘制以及硬件加速方法；

（3）给出本章基于动态纹理的角色绘制算法思想的实现方法，并通过运动角色仿真实验验证算法有效性；

（4）扩展本技术的使用范围，介绍利用本章的运动角色仿真技术，实现大规模动态森林场景仿真的方法。

## 6.1 相关工作

交互性是虚拟现实的基本特征之一，其最重要的功能是操作者可以感受到场景的动态变化并进行交互控制。在大规模运动角色仿真中，影响仿真交互性的因素有两方面，一方面是仿真中的角色运动的变形计算速度，另一方面是角色的绘制速度。变形计算速度依赖于变形算法的改进，而绘制速度则依赖于对场景中角色的模型简化以及硬件加速等来实现。为此，研究者们提出了多种加速计算技术和加速绘制技术，用来提高图形系统对模型的处理能力和绘制能力，其中对于场景中的模型简化和快速绘制能力的提高，常用的有 LOD 层次细节和基于图像的渲染两种方法。

LOD 技术通过减少需绘制的多边形数量，提高模型绘制的效率。目前对于大量模型或角色的绘制，如队形编排等，一般采用 LOD 模型简化方法[17]加快绘制速度。LOD 方法将一个模型使用多个三角形分辨率层次表达，在模型渲染中，根据模型对屏幕的贡献大小选择不同的层次。Heok[17]给出了一个 LOD 方法的总结，潘[89]等人给出了一个多细节层次模型自动生成技术的综述，Hoppe[91][92]提供了一种模型简化与渐进式网格简化方法，Cignoni[93]使用 LOD 方法进行模型的简化和压缩，Ogren[94]使用连续 LOD 方法实现实时的地形渲染，张昌明等人[95]提出了一种基于边折叠的多边形网格模型简化算法，能在损失很少的屏幕像素误差的前提下提高图形绘制速度。

LOD 方法对提高绘制效率的效果十分明显，对减少存储需求、减少计算复杂度以及对网络模型传输也都有重要的意义，但许多模型并不适合 LOD 简化，如树木模型。另外，LOD 简化使模型真实感降低，而且其进行角色绘制时所支持的规模大约为 100 人，仍然不能实现上千规模的角色同时实时计算和渲染[124]。

基于图像的绘制方法[19]是基于几何方法的一种功能强大的替代方法，它通过一个图像集合和相应的深度图来表示一个场景或者模型，绘制速度与模型的复杂度无关，十分有利于复杂物体的实时绘制。基于图像的绘制大体分为基于无几何、隐式几何和准确几何三类方法，传统的纹理映射方法，就是一种依赖于准确

的几何模型但只需要少量图像的方法，天空绘制一般采用无几何的全景图方法，而 Billboard 方法则是一种常用的基于隐式几何的绘制方法。

Billboard 是一种很常用的模型渲染技术[125]和复杂模型简化技术[126]，它将模型投影图像作为纹理，以朝向观察者的平面板代替模型，能够很好地实现大量复杂模型的高速渲染。Billboard 技术与 Alpha-Texture 技术结合，能够显示多种难以用实体造型描述的物体，如烟、雾、火、爆炸、云[98,99,100]、树木等。但 Billboard 与 Alpha-Texture 的结合只能将模型的一个姿态投影成静态的图像，不能在三维场景中展现模型的动态效果，这样很大地限制了它的应用范围。

基于图像渲染技术的动态景物渲染，一般需要进行重渲染或者纹理更新来实现。Schaufler[101,102]通过每隔几帧重新渲染原多边形模型，从而更新 Billboard 的纹理，但每隔几帧重新渲染大量多边形需要消耗很多时间。Max[104]预计算若干个离散视点的纹理图像，然后通过插值生成各视点的纹理，实现树木模型的动态效果。

本章针对大规模运动角色的实时绘制问题，使用基于图像的渲染技术、Billboard 技术以及图形硬件加速技术，实现大规模运动角色的实时变形和渲染。与其他类似方法相比，本章方法的主要不同之处在于：

（1）使用 Billboard 作为运动角色的载体，使用基于图像的绘制代替模型绘制，保证大量模型的渲染速度；

（2）通过渲染到纹理技术将变形结果保存为动态纹理，并在运行时同步更新到 Billboard 的纹理中，实现 Billboard 所展示内容可以实时动态改变；

（3）使用纹理复用降低场景中的原始模型数量，并由此减少图像变形中的计算量；

（4）使用 GPGPU 技术加速图像变形计算和场景绘制，进一步提高运行效率，保证大规模模型同时运动仿真的实时性。

## 6.2　复杂模型快速绘制技术

实时三维应用对计算机的系统需求很高，既要求绘制的真实性，又要求绘制的实时性，同时需要在保证一定渲染效果的条件下同时保证一定的交互帧速率，因此即使是在配备高性能的图形工作站，对于复杂的场景渲染，仍然需要绘制技术的优化。

计算机能够达到实时绘制取决于两点：第一点，所采用的计算机平台具有很高的运算速度；第二点，所绘制的目标模型要具有尽量少的面片数。在固定帧速率条件下，一台计算机所能绘制的原始几何面片数是有限的，因此简化绘制目标的复杂度是实现复杂模型实时绘制的主要途径，即通过减小计算量来实现实时性

能。针对虚拟场景的实时与真实感绘制，研究人员们提出了大量的加速技术，包括运动预测、可视化剔除、LOD、场景分割等。在这些技术中，常用的模型简化方法有 LOD（Level of Details）简化技术和基于图像的渲染技术（Image-Based Rendering，IBR）。

### 6.2.1 LOD 绘制技术

现代三维扫描技术的发展使得三维模型数据呈现高速的增长状态，与此同时，模型的大小同样增长迅速，上 G 大小的模型已经非常常见。随着图形学硬件性能的提高，现代计算机系统的图形渲染能力随之提高，但仍然无法满足当前艺术级的图形渲染能力。因此在渲染过程中的实时模型简化方法会被大量应用，这些模型简化方法的最典型代表就是 LOD 技术。

LOD 技术将一个模型简化为多个不同的三角形分辨率层次，每个层次包含不同的细节程度，在实际绘制过程中，根据对视觉的贡献率选择不同的分辨率层次，当该模型在场景中的重要性降低时，选择较低分辨率的模型表示层次，当离观察者较近或非常重要时，使用更高分辨率的模型。图 6-1 所示为 LOD 方法保存的不同分辨率的模型。

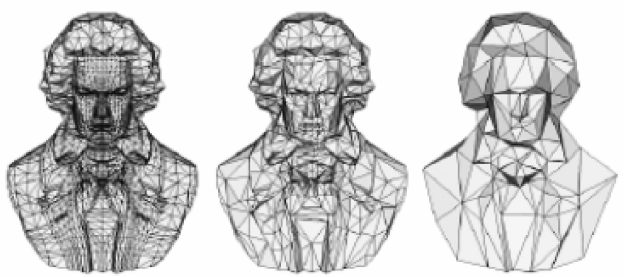


图 6-1 LOD 方法保存的不同分辨率的模型

目前 LOD 技术中，有离散 LOD、连续 LOD、视点相关的 LOD 和层次 LOD 几种[17]。离散 LOD 是传统的 LOD 方法，它将模型预处理为互相独立的细节层次模型，在运行时根据该模型对整个场景的贡献率选择不同的 LOD 层次，因此称为离散 LOD。离散 LOD 的最显著的优势在于编程实现较为容易，而且该方法适用于所用图形加速硬件。离散 LOD 方法可以被编译成三角带、显示列表、顶点数组等，进一步加速显示性能。但离散 LOD 的缺陷是无法应用于一些形状特殊组织结构的模型简化，对于大模型的简化效果并不好。

连续 LOD 在 1976 年提出，相对于传统的离散 LOD 方法，连续 LOD 在运行时提取 LOD 细节，使得简化在运行时进行，根据当前所需的细节程度，准确地简化出一个特定的层次细节，而离散 LOD 是选择一个近似的预创建的模型。在

相同绘制效果条件下，连续 LOD 使用更少的三角形，使得系统绘制性能进一步提高。连续 LOD 的另一个优点是可以实现层次间的平滑过渡，减少了 pops 的出现。

视点相关的 LOD 方法针对当前的视点位置和视线方向，选择最佳的层次表示。在显示过程中，同一个模型的不同部分可能使用不同的 LOD 细节层次，距离视点越近的部分使用越高的分辨率，而较远的部分使用较低的分辨率。侧影轮廓部分使用更高的分辨率，而正对视点的位置使用较低的分辨率。在同样三角形数目的条件下，视点相关的 LOD 方法比连续 LOD 具有更好的精细度，这是因为视点相关的 LOD 方法将多边形分配在最需要的位置上。视点相关的 LOD 对超大模型的简化效果要更好。

层次 LOD 是针对于大量小模型的显示问题而设计，在模型重要性较低时，将若干个小物体组合为一个整体进行简化而不是对每个物体进行简化，解决了小模型的显示问题。层次 LOD 与视点相关 LOD 具有同样的稳定性，将整个场景看作一个整体进行简化。

在 LOD 方法的使用中，Heok[17] 和何晖光[88] 分别给出了一个 LOD 方法的综述和网格模型简化的综述，潘志庚[89] 给出了多细节层次模型自动生成技术的综述，Chadwick[90] 给出了一种动画角色的层次化创建方法，Hoppe[91,92] 提出了一种不规则网格的离散简化与渐进式简化方法，Cignoni[93] 比较了各种网格简化算法的优劣，Ogren[94] 提出了一种基于连续 LOD 的实时地形渲染算法，2005 年张昌明等[95] 提出了一种基于边折叠的多边形网格模型简化算法，在减少屏幕误差的前提下提高图形绘制速度。

LOD 简化过程可以分为多边形简化和非多边形简化，对于非多边形简化包括样条曲面简化、体模型简化以及基于图像的模型的简化。大多数 LOD 简化算法是针对多边形简化的。多边形 LOD 简化算法可以分为两部分，一部分是几何简化，另一部分是拓扑简化。几何简化主要是减少几何元素（顶点、边、三角形等）的数目，拓扑简化主要为了减少孔洞的数目。一些复杂的简化算法常常既包含几何简化又包含拓扑简化。

几何简化大体可以分为如下几类：

（1）顶点剔除。该方法主要是减少模型中的“Pluck”顶点，每剔除一个顶点，其周围的相邻三角形就同时被删除，同时通过重新三角化补齐由于删除顶点造成的孔洞。顶点剔除方法可以应用于所有的多边形模型。

（2）顶点合并。顶点合并法将相邻的顶点进行合并，重新计算一个合适的顶点替代这些顶点及其被合并的三角形，简化之后密集区域的顶点和三角形被合并成一个顶点。

（3）边删除。在边删除算法中，一条边两端的顶点合并为一个顶点，同时删

除该边及其两侧的三角形。该方法的优点是替代顶点的位置可以自由选择，而且不需要执行较为复杂的区域三角化算法。通用的边删除算法包含两个步骤：一个是选择需要合并的边，通常是根据视觉差来决定的，二是计算替代顶点的位置。

（4）顶点对收缩。顶点对收缩是一个比边删除更加稳定的算法，该算法支持任意的一对顶点进行合并，不论它们是否有共享边。

（5）面合并。面合并算法支持共面面片的合并，从而形成“超面”。在合并过程中，整个网格通过聚簇算法分为几个区域，区域内的内部顶点全部删除，剩下的顶点形成一个简化的网格模型，最后将模型中非平面的“超面”进行三角化即可。

### 6.2.2 基于图像的绘制技术

基于图像的绘制技术是应用于计算机视觉和计算机图形学两个领域的通用技术。传统的创建虚拟场景的方法是使用 3D 模型直接进行渲染，而使用 CAD 模型获取的数据一般是多边形、双线性插值面片、连续实体几何以及八叉树表示方法。一般的场景还需要通过纹理映射、环境映射、阴影算法增强场景的真实性。

基于图像的渲染技术与传统的 3D 模型渲染方法的区别见表 6-1。

**表 6-1 3D 模型的绘制与图像绘制的区别**

| 基于 3D 模型的绘制 | 基于图像的绘制 |
|---|---|
| 使用 3D 模型 | 直接使用图像集合 |
| 使用传统渲染流程 | 基于插值或像素映射 |
| 绘制速度依赖于场景复杂度 | 绘制速度与场景复杂度无关 |
| 绘制速度依赖于 GPU 速度 | 绘制速度依赖于 CPU 速度 |
| 通过软件过程完成渲染 | 直接通过输入图像完成 |

在传统的 3D 绘制技术中，所使用的模型是表达准确的 3D 模型，场景的更新通过这些模型的位置改变或者观察点位置改变来实现。而基于图像的绘制技术中使用的是所表示物体的图像集合，真实物体和场景的图像获取相对更容易些。

3D 渲染技术通过传统的渲染通道实现，包括模型变换、视图变换、背面剔除、消隐等步骤，整个渲染过程的复杂度取决于模型的面片数或体元数。为渲染复杂场景，一般来说需要使用加速算法。与使用 3D 模型创建虚拟场景相比，基于图像的绘制技术仅仅使用原始的图像即可创建新的虚拟场景。基于图像的绘制技术通过输入图像的数据集中选取源图像和目标图像，然后对前后的图像进行插值或者进行体素映射，从而生成新的虚拟场景。渲染过程的复杂度与场景的复杂度无关，而且由于避免了大量的计算，不需要特定的硬件加速方法。

根据渲染对象的承载体，基于图像的绘制技术[19,96,97]又分为无几何、基于隐式几何和基于准确几何三类。无几何图像渲染使用全景图来构建虚拟环境，在虚

拟环境中漫游相当于选择不同的全景图。基于 Billboard 的渲染方法则是一种基于隐式几何的图像渲染方法。而传统的纹理映射方法是一种依赖于准确的几何模型但只需要少量图像的方法。图像渲染方法的优越性是生成图像的质量与场景的复杂性无关，不用专门的硬件加速就能获得很强的真实感和实时的交互速度。

根据像素索引方式的特性，基于图像的绘制技术也可以分为四类：非真实感图像映射、基于图像的镶嵌技术、密集采样插值技术以及几何近似的像素映射技术。

（1）非真实感图像映射将两幅图像由源图像缓慢过渡到目标图像中，在该类技术中，像素位置计算中不考虑 3D 几何模型，非真实感图像映射通过不相关的两幅图像实现插值映射。这种技术广泛地应用于广告、娱乐业。

（2）基于图像的镶嵌技术将两幅图像结合成一幅更大或者分辨率更高的图像。早期的图像镶嵌技术常用于地球卫星图片和大气数据的合成等。

（3）密集采样插值技术是通过围绕一个物体或者一个场景，从各个不同视点拍摄一系列图片，然后通过插值这些事先拍摄的图片从而生成任意视点上的场景。这种方法的优势在于不需要其他方法中所需的复杂的像素映射过程。

（4）几何近似的像素映射技术使用相对较少数量的图片，在一个指定的摄像机视点位置上，重新映射图像像素。几何模型的约束规则可通过像素的深度来给出。当一个像素的深度已经给出之后，相应地可以计算像素所对应的位置。

## 6.3　基于动态纹理的运动角色绘制技术

### 6.3.1　大规模动态角色绘制的特点

运动角色仿真与其他应用中的角色仿真具有共同的要求，那就是对真实感的要求，这也是计算机场景仿真中的普遍要求。对于运动角色的仿真，其模型细节表达程度、运动动作流畅性和场景绘制速度是三个影响真实感的主要因素。

模型的细节表达程度是决定仿真结果真实感的一大因素。常用的 LOD 简化方法将原始模型进行简化，生成粗细不同的多个模型层次，在不同的情况下选择不同的模型层次。假如选用的模型层次过于简化，就会在角色的某些部位出现严重的失真。影响角色仿真的另一大因素是运动动作的流畅性和连贯性。对于运动角色，动作控制中很容易出现动作不连贯的不自然现象，从而影响真实感。对角色仿真的真实感影响较大的最后一个因素是运动角色的绘制速率，帧速率是影响所有虚拟现实系统真实感的重要数据，在计算机实时仿真中，一般需要达到 30FPS 的帧速率。

运动角色仿真又有许多其独有的特点：

（1）因为角色众多，不可能进行人工交互，必须对各角色事先设定动作数据，并自动变形；

（2）运动场景中的角色仿真，需要考虑动作统一性的要求，对于大规模场景排演，所有角色必须按照统一的动作要求进行，以达到整齐划一的效果；

（3）对于大规模角色场景，虽然角色数量大，但当可视化距离足够远时，场景中的角色会出现很大的重复性。根据该重复性特点，可以考虑使用一些复用手段。

运动场景中的角色仿真需要解决一些新出现的问题：

（1）角色的自动动作驱动方式，该动作驱动方式，应该拥有不依赖于模型的驱动方法；

（2）需要实现角色运动的分解，为避免重复计算，充分利用场景中角色重复性特点，将角色的运动分解为以自身为中心的动作和其在整个场景中的位置变化，这样就可以针对角色模型本身进行变形计算，同一计算结果可多次应用到场景中的角色上。

### 6.3.2 动态纹理技术的原理

在传统的渲染流水线中，每次渲染运算的最终目的地就是帧缓冲区，然后实时地输出到屏幕显示出来。图形硬件的发展使得图形硬件支持离屏渲染，渲染到纹理技术[120]允许将一个普通 OpenGL 纹理绑定到 FBO 对象上，并设置 FBO 为渲染对象，从而将渲染计算的结果保存到 FBO 绑定的纹理中。通过纹理操作可以进行显存和内存之间的数据交换，这也是 GPGPU 技术[108]返回计算结果的方法之一。

动态纹理顾名思义是一种可以在运行时进行实时内容更新的纹理，在本章中所指的是利用图像硬件支持，在显存中生成的一个作为渲染目标的纹理。与通过上传数据创建的纹理相比，本章所指的动态纹理直接在 GPGPU 计算过程中产生，并直接在显存中参与渲染过程，不需要从内存中上传纹理数据或往内存回传数据，节省了数据交换的时间，可以用于高速的渲染过程中。

在纹理 ID 与纹理坐标都不改变的情况下，可以在运行时更新目标模型上的纹理内容，动态纹理的内容是在运行时动态改变的。动态纹理更新过程可以描述为：

（1）将纹理绑定到 FBO 对象上，设置 FBO 绑定纹理为渲染目标；

（2）通过渲染到纹理技术，将一个过程的渲染结果输出到 FBO 对象绑定的纹理中，从而实现纹理内容的动态更新。

传统的 Billboard 方法使用静态图像作为纹理，只能仿真静止不动的角色，而在运动角色仿真应用中角色是一直活动的，无法使用 Billboard 实现活动角色的动

态渲染，只能使用 LOD 技术简化 3D 模型，加速渲染速度。Billboard 与动态纹理的结合，可以有效解决 Billboard 技术只能渲染静态图像的问题，提供了一种快速渲染场景中的大量动态角色的方法，这种方法既保持了 Billboard 技术的快速渲染特性，又引入了所绘制角色的动态特性。显存动态纹理因为不涉及显存与内存的数据交换，在显存中直接应用于场景绘制过程，运行时间很短，对渲染进程几乎没有影响。

### 6.3.3　运动角色的动态纹理创建

借助于动态纹理技术，将运动角色的当前变形结果，渲染为可以实时更新的动态纹理，从而在 Billboard 上实现可以活动的角色仿真，动态纹理是将单个角色的变形用于大规模运动角色绘制的关键，动态纹理需要在每一次的角色变形时更新一次内容。

动态纹理的创建过程如图 6-2 所示，渲染内容通过图中所示的传送次序，最终使角色图像变形渲染的结果保存到显存空间的动态纹理中。

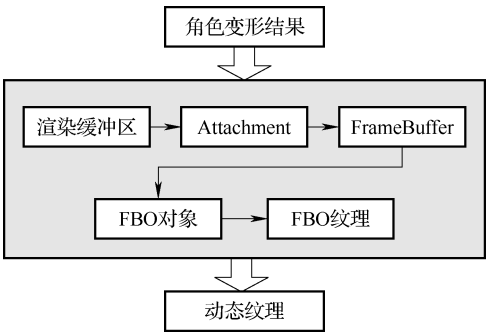



图 6-2　动态纹理的创建过程示意图

FBO 纹理的绑定过程可描述为

（1）在显存中申请离屏渲染对象 FBO，申请指定大小的纹理空间；

（2）将该纹理和 FBO 对象绑定，纹理大小要与渲染视窗一致，在申请方式上与普通的纹理相同；

（3）将 FBO 对象绑定到当前的 FrameBuffer 对象上；

（4）设定 FrameBuffer 的数据来源，如 Attachment0；

（5）设置该 Attachment 为渲染目标。

在动态纹理的使用过程中，设置 FBO 对象为渲染目标，渲染当前动作状态的角色变形结果，渲染完毕后，动态纹理的内容就已经更新。动态纹理更新之后，重新设置帧缓冲区为渲染目标，即可使用该纹理进行实际的屏幕渲染。

### 6.3.4 基于动态纹理的角色绘制

基于 Billboard 的图像渲染有两种数据需要计算，一种是 Billboard 的顶点坐标，另一种是 Billboard 的纹理坐标。根据 Billboard 位置计算其顶点坐标的方法如图 6-3 所示，点 $P$ 为眼坐标位置，四边形 $ABCD$ 表示 Billboard 面板，该面板朝向眼位置，其底线 $AB \perp PR$。$ABCD$ 四个顶点的坐标计算方法为：

（1）根据 Billboard 的宽度和所处位置 $R$ 计算水平线 $EF$ 的两点坐标；

（2）根据视方向角 $\alpha$ 计算底线上的两顶点 $AB$ 的坐标；

（3）根据 Billboard 的高计算 $CD$ 两顶点的坐标。

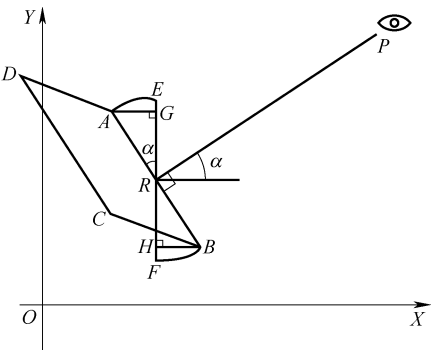



图 6-3 Billboard 顶点坐标计算示意图

因为绑定到 FBO 的纹理是规则纹理，而角色图像一般不能与纹理正好大小相符，只有动态纹理的一部分是角色图像的有效区域。如图 6-4 所示为有效区域纹理坐标计算示意图，其中外框 $ABCD$ 表示 FBO 绑定纹理的有效大小，$EFGH$ 为运动角色的有效范围，亦即将最终贴图到角色替代 Billboard 上的内容部分。以点 $G$ 为例，该点将贴图到 Billboard 的右下顶点，该顶点的纹理坐标可由式（6-1）计算得出。

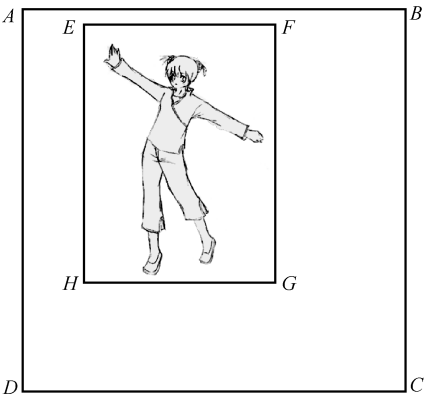



图 6-4 动态纹理坐标计算示意图

$$\begin{cases} x = X_G / X_{FBO} \\ y = (1 - Y_G) / Y_{FBO} \end{cases} \tag{6-1}$$

依次计算各角色的替代 Billboard，依据其位置坐标和动态纹理的有效区域纹理坐标，将由运动角色生成的动态纹理贴图到 Billboard 上，并渲染到三维场景中，最终生成所期望的大规模运动角色场景。

### 6.3.5 运动角色绘制的硬件加速

在运动角色场景仿真应用中，有两个过程可以应用 GPGPU 技术加速计算，分别是运动角色变形过程和运动角色绘制过程。对于二维角色变形，可以使用硬件加速计算自适应变形网格顶点坐标，从而减少角色变形的计算时间。在运动角色绘制过程中，可以使用硬件加速计算 Billboard 的顶点坐标。对于二维角色变形

的硬件加速，在本书第 2 章中已经详细介绍 GPGPU 加速的变形网格计算的两种方法。本节介绍采用 GPU 顶点渲染器加速计算 Billboard 顶点坐标的方法。

根据该角色的运动轨迹，计算角色在场景中该时刻的位置，然后将该位置传递给 GPU 顶点程序，在 GPU 中计算。在顶点程序中 Billboard 的顶点坐标，根据观察者所在位置和角色高度宽度，计算 Billboard 顶点坐标。角色绘制加速计算的 GPU 程序输入是观察者的位置、各活动角色的高度宽度及其在三维场景中的位置，输出是场景中各活动角色 Billboard 的顶点坐标，在三维场景中应用角色变形生成的动态纹理渲染 Billboard，最终得到三维角色运动场景。

GPU 加速角色绘制的顶点程序可以描述为

（1）计算从观察者到活动角色的视向量；

（2）计算当前角色 Billboard 板的旋转角度；

（3）计算 Billboard 板顶点在局部坐标系中的坐标；

（4）根据活动角色位置，把局部坐标转换成全局坐标。

## 6.4　基于动态纹理的角色绘制算法实现

本章针对大量动态角色的实时绘制问题，提出一种基于动态纹理的快速绘制方法，通过 Billboard 技术和动态纹理技术的结合，在保持 Billboard 渲染技术的高效性的同时，实现了运动角色的动态渲染。图 6-5 所示为运动角色绘制的数据流图。

本章基于动态纹理的角色绘制算法实现可以总结为：

（1）初始化过程。加载角色动作驱动信息，计算关键帧动作状态数据；在显存中申请纹理空间和离屏渲染对象 FBO，并绑定纹理到 FBO 上；设定场景参数、观察者位置等参数；加载三维场景，计算各运动角色位置；加载图像变形 GPU 顶点计算程序；加载场景渲染计算 GPGPU 顶点程序。

（2）每一帧的渲染过程。首先，根据当前程序运行时间，查找前后关键动作状态，计算当前时刻的关节点数据；然后，设置渲染对象为 FBO 离屏缓冲区，启用图像变形 GPGPU 顶点程序，计算角色图像变形网格，完成图像变形并生成动态纹理；最后，重置渲染对象为屏幕缓冲

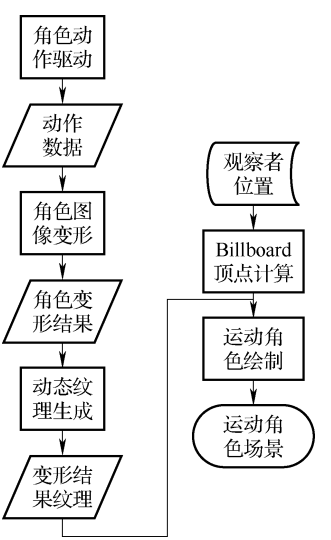



图 6-5　运动角色绘制的数据流图

区，启用场景渲染 Billboard 顶点计算 GPGPU 顶点程序，渲染三维角色运动场景。

## 6.5 大规模角色群组绘制关键代码

本章的大规模角色群组绘制功能，使用 Billboard 代替大量的角色进行绘制，使用动态纹理技术将运动角色的运动过程实时绘制为动态纹理，并映射到 Billboard 板上，使 Billboard 的绘制既保持了快速绘制特性，又引入了动态特性，从而实现大量群组运动角色的快速绘制。

在本章的大规模角色群组绘制过程中，除使用前面几章中的动作驱动代码、群组运动角色变形代码等，还需要使用 Billboard 绘制，以及使用动态角色变形的 GPU 加速和 Billboard 渲染的 GPU 加速程序，最后还在本章中引入计时功能。

### 6.5.1 从立点扩展成 Billboard 板的顶点

使用 Billboard 进行绘制变形角色，首先需要根据 Billboard 所在位置扩展成 Billboard 板的四个顶点的坐标。位置向 Billboard 顶点扩展的代码如下：

```
void arrangeBBVexData(float * * BBstandP,int BBnum1)
{
    //x 和 z 变化，y 其实不变化的，可以用作板上的顶点序号
    BBnum = BBnum1;
    //BBnum 个板，每个板 4 个顶点，每个顶点 4 个坐标值
    delete [ ] BBVertex4;
    BBVertex4 = new float [16* BBnum +1];
    ASSERT (BBVertex4! = NULL);
    //输入坐标 (standX, standZ, left0/right1, bottom0/top1);
    static float crdT[16] = {
        0, 0, 0, 0,
        0, 0, 1, 0,
        0, 0, 1, 1,
        0, 0, 0, 1,
    };
    float * pBBs = * BBstandP;
    float * pBBv = BBVertex4;
    for (int i = 0; i < BBnum; i + + )
    {
       memcpy_s(pBBv,16* sizeof(float),crdT,16* sizeof(float));
```

```
            * (pBBv) = * (pBBv + 4) = * (pBBv + 8) = * (pBBv + 12) =
            * (pBBs + + );
            * (pBBv + 1) = * (pBBv + 5) = * (pBBv + 9) = * (pBBv + 13) =
            * (pBBs + + );
            pBBv + = 16;
        }
    }
```

## 6.5.2　动态角色变形计算 GPU 加速程序

群组运动角色的变形计算与 Billboard 板顶点的坐标计算是本群组仿真应用中计算量最大的部分，因此在本书的仿真应用中使用 GPU 实现这两部分运算的加速。其中动态角色变形技术的 GPU 加速程序代码如下：

```
void DeformVex (
    in float4 pos4i: POSITION,
    in float4 color4i: COLOR0,
    in float4texcrd4i: TEXCOORD0,
    in float3 normal3i: NORMAL,
    out float4 pos4o: POSITION,
    sout float4 color4o: COLOR0,
    out float2 texcrd2o: TEXCOORD0,
    uniform float ptDivNum1,
    uniform float shape1,
    uniform sampler2D dataTex,
    uniform float dTexSize,
    uniform float4x4 ModelViewProj//这里最后一个不能有逗号
)
{
    //pos4i 是 Length,Width,SerialNum,curjpt.pt
    float4 ptDA = texcrd4i;
    //提取变量
    float lenC = pos4i.x;
    float widC = pos4i.y;
    float alpha1 = radians (ptDA.w/2); //转为弧度
    if (pos4i.z > 0 && alpha1 > 0) widC = - widC;
    //求 Yend
```

```
    float sinV, cosV;
    sincos(alpha1,sinV,cosV);
    float signW = (lenC + 1) * alpha1;
    float yend = - lenC* sinV + abs(signW)/signW* widC* cosV;
    float xCur, yCur;
    if(pos4i.z < 0)
    {
        //矩形部分
        yCur = yend;
        xCur = abs(lenC)* cosV - widC* sinV;
    }
    else if(yend* lenC* alpha1 > 0){
        yCur = 0;
        xCur = abs(lenC)* cosV - widC* sinV;
    }
    else
    {
        //曲线部分
        float aC = abs(widC* shape1/sinV);
        float bC = abs(widC* shape1/cosV);
        float tX = aC* sqrt(pow(lenC* sinV/bC,2) + 1) - abs(lenC* cosV);
        yCur = yend* pos4i.z/ptDivNum1;
        xCur = aC* sqrt(pow(yCur/bC,2) + 1) - tX + widC/abs(sinV);
    }
    sincos(radians(ptDA.z + ptDA.w/2),sinV,cosV);
    float4 posC;
    posC.x = xCur* cosV - yCur* sinV + ptDA.x;
    posC.y = xCur* sinV + yCur* cosV + ptDA.y;
    posC.zw = float2 (0, 1);
    pos4o = mul(ModelViewProj,posC);
    texcrd2o = float2(normal3i.x/1024,1 - normal3i.y/1024);
    color4o = color4i;//float4(1,1,1,1);
}
```

### 6.5.3　Billboard 板顶点坐标计算 GPU 加速程序

Billboard 板的顶点坐标计算程序代码如下：

```
void cgScVexP(
    in float4 pos4i: POSITION,
    in float4 color4i: COLOR0,
    in float2 texcrd2i: TEXCOORD0,
    out float4 pos4o: POSITION,
    out float4 color4o: COLOR0,
    out float2 texcrd2o: TEXCOORD0,
    //这里输出 POSITION 是 float3/float4 都可
    uniform float3 eyePos3,
    uniform float2 hwBB2,
    uniform float4x4 ModelViewProj
)
{
    //输入坐标 (standX, standZ, left0/right1, bottom0/top1);
    //求眼点至立点的向量，并根据这个向量，对四方框进行旋转
    float2 vectEye2BB2 = pos4i.xy - eyePos3.xz;
    float2 vectRot2 = float2(-vectEye2BB2.y,vectEye2BB2.x)/
          length (vectEye2BB2);
    //旋转：四方框坐标坐标 (+/-0.5* width, 0)
    //因 y==0，因此 x' = x* (-dy/r); y' = x* (dx/r);
    float2 localRotCrd2 = vectRot2* hwBB2.y* (pos4i.z - 0.5);
    //移动坐标至立点
    localRotCrd2 += pos4i.xy;
    float4 posCrd4 = float4(localRotCrd2.x,hwBB2.x* pos4i.w,
          localRotCrd2.y, 1);
    //场景视景变换
    pos4o = mul (ModelViewProj, posCrd4);
    color4o = color4i;
    texcrd2o = texcrd2i;
}
```

### 6.5.4 程序运行计时功能函数代码

在本书的大规模角色群组仿真过程中，为测试整个系统的计算性能，在系统中添加程序运行的计时功能。本系统中所带的计时功能，使用 Windows 操作系统 PlatformSDK 开发包的 QueryPerformanceCounter（）函数，通过计数 CPU 计算周期从而获得最精确的计时。该计时方法可以实现 ns 级的计时准确性。在系统不支持该计时方式时，还提供 GetTickCount（）这种基本计时方法。

**1. 计时功能初始化部分**

计时功能函数分为初始化、计数和计算几部分。其中初始化部分主要是获取计时周期、开辟内存存储空间等，其代码如下：

```
bool initTimeStat(int recNum1,int vectSize1,int recMode1)
{
    //初始化，设定记录途径，获取基本参数，申请内存空间
    //获取时间模式，QueryPerformance (1), getTick (else)
    allSuffix = recNum1;
    vectSize = vectSize1;
    recMode = recMode1;
    if (recMode ==1)
    {
        //QueryPerformance 方式记录时间
        //取得 CPU 频率：
        QueryPerformanceFrequency(&frequency);
        performRatio =1000.0f/(float)(frequency.QuadPart);
        PerformCount =new LARGE_INTEGER [allSuffix* vectSize];
        ASSERT (PerformCount! =NULL);
        memset (PerformCount, 0, allSuffix* vectSize*
                sizeof (LARGE_INTEGER));
    }
    else
    {
        tickCount =new DWORD [allSuffix* vectSize];
        ASSERT (tickCount! =NULL);
        memset (tickCount, 0, allSuffix* vectSize* sizeof (DWORD));
    }
    allSuffix - -;
```

```
    return true;
}
```

**2. 计时功能计数记录部分**

计时功能函数的计数记录部分主要功能是记录当前数据，并存储在向量的第 *N* 分量中，代码如下：

```
void recValue(int vectComponent)
{
    glFinish();
    if(recMode ==1)
    {
        //QueryPerformanceCounter(&start);
         QueryPerformanceCounter(PerformCount + (curSuffix*
                vectSize + vectComponent));
    }
    else
    {
        * (tickCount + (curSuffix* vectSize + vectComponent)) =
                GetTickCount();
    }
}
```

**3. 计时功能时间计算部分**

本部分是根据初始化过程中获取的计时间隔（即计时性能），以及计时开始与结束时的计数计算所经过的时间，计算部分示例代码如下：

```
bool calExample1()
{
    //计算渲染时间
    if (recMode ==1)
    {
        LARGE_INTEGER curPCount, startPCount;
        memset(&startPCount,0,sizeof(LARGE_INTEGER));
        curPCount = startPCount;
        float renderTime = (float)(curPCount.QuadPart -
                startPCount.QuadPart)/(float)(frequency.
                QuadPart)* 1000;
    }
```

```
    else
    {
        DWORD curTCount = 0, startTCount = 0;
        float renderTime = (float)(curTCount - startTCount);
    }
    //renderTime 毫秒数
    return true;
}
```

## 6.6 本章小结

本章针对大规模运动角色绘制中的实时性问题，提出了一种基于动态纹理的快速绘制大规模运动角色场景的方法。该方法通过渲染到纹理技术，将变形结果保存为显存动态纹理，有效地解决了 Billboard 技术只能渲染静态图像的问题。使用基于动作迁移的角色驱动方法实现运动角色的自动驱动，并有效地避免了角色动作配置繁琐的问题；使用基于自适应网格的二维角色变形方法，实现了运动角色的平滑变形；使用 GPGPU 技术加速计算二维角色变形中的自适应网格顶点和 Billboard 复合纹理坐标，提高场景渲染速度。

通过大规模运动角色的仿真实验，验证了方法的可行性，并对仿真方法的使用范围进行扩展，用于大规模动态森林场景的绘制。

进一步的工作将致力于研究运动角色的三维运动快速可视化方法，以及含有多角色交互的大规模运动角色场景绘制方法。

# 第 7 章　大规模角色群组场景仿真方案

结合本书的大规模角色变形方法、角色驱动方法和角色快速绘制方法，可以实现大规模运动角色的仿真，该仿真过程由三部分组成，分别是角色动作驱动、角色模型变形和角色快速绘制，其中角色动作驱动和角色图像变形共同实现单个运动角色的仿真，通过角色快速绘制技术，将单个角色的仿真方法应用于大规模角色的仿真。图7-1所示为大规模角色群组仿真流程图。

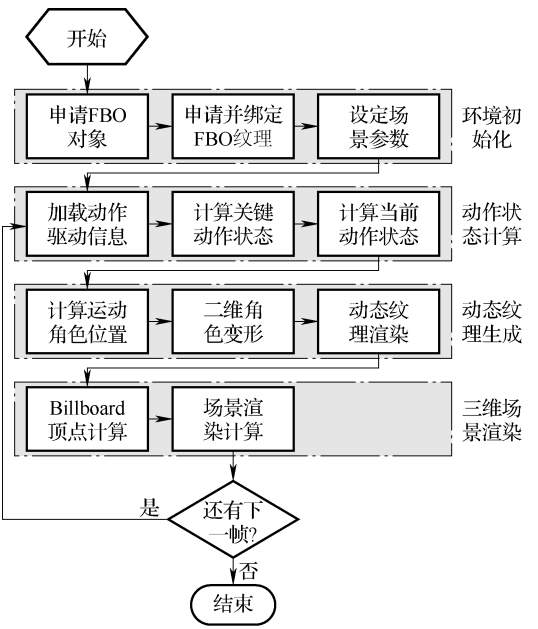



图 7-1　大规模角色群组仿真流程图

角色动作驱动部分根据预先设定的动作计算场景中各运动角色的当前动作状态，该部分使用基于视频的动作驱动方法更新角色动作状态。动作驱动过程中，加载动作数据，在运动过程中改变每个关节点的旋转角，根据骨骼框架信息计算各关节点的新位置，实现虚拟角色的动作驱动。

角色变形部分负责根据运动角色的当前动作状态计算其变形结果。该部分使用基于自适应网格的图像变形计算，在角色变形过程中，根据动作驱动步骤中获得的各关节点新位置，更新各关节点的自适应网格，获得对应于该动作状态的角色变形结果。

角色快速绘制部分结合Billboard技术和动态纹理技术快速绘制运动角色，渲染到纹理技术将运动角色的当前变形结果渲染为可以内容实时更新的动态纹理；最终获得大规模运动角色的动态场景。

基于本书的思想与方法，现已经在 PC 上实现了一个基于动态纹理的大规模运动角色快速绘制演示程序。本章实验的硬件配置为 Intel P4 3.0G CPU，1GB 内存，nVidia GeForce FX6600 128M 显卡，软件环境为 Windows XP 操作系统，编程

环境为 Microsoft Visual Studio 2005，3D 开发环境为 OpenGL。

本节的实验使用的角色选为 Girl 角色，在实验过程中，实现该角色的运动场景绘制。通过实验结果及数据的分析，验证本书提出的角色群组仿真方法的有效性。

本节的实验主要包含如下几部分：

（1）验证通过动态纹理技术实现大规模运动角色快速绘制方法的可行性。本书提出使用结合动态纹理技术和 Billboard 技术的方法，实现大量角色的快速绘制，并结合本书提出的运动角色变形方法和角色动作驱动方法，共同完成大规模运动角色的实时仿真。通过演示系统的绘制结果，说明该方法的可行性和有效性。

（2）获得动态场景仿真速度与运动角色数量的关系。本章提出的快速绘制方法及角色变形方法，主要是针对于大量角色的快速仿真应用，因此分别使用这些方法所实现的动态场景中，能实时仿真的运动角色的数量是一个重要的衡量标准。通过分别统计不同角色数量下的各步骤执行时间，获得运动角色数量与仿真速度的关系，并同时验证图形硬件加速的有效性。

（3）获得动态场景仿真运行时间与角色种类数的关系。本书提出的大规模运动角色快速绘制方法得益于两个方面：一方面是基于 Billboard 绘制的快速性和角色变形算法的快速性，另一方面是在实际绘制过程中使用了纹理复用技术，使用同一个纹理内容进行绘制类似的角色。场景中不同角色的种类数直接影响场景中的多样性，并引起角色变形过程的运行时间变化。通过分别统计不同变形角色种类数下的运行时间，说明角色种类数对执行时间的影响。

## 7.1 基于动态纹理的大规模角色群组仿真结果

本书提出的大规模角色群组仿真方法，结合 Billboard 技术和动态纹理技术共同实现场景中运动角色的快速绘制。结合本文前几章提出的图像变形方法和动作驱动方法，可以高效地仿真大规模角色群组场景。在本实验中通过使用 Girl 角色绘制大规模角色群组场景，验证该快速绘制方法的可行性及有效性。

**实验：Girl 角色的仿真结果**

本章的实验是以 Girl 作为原始角色，按照本章的方法对 5 × 5 个 Girl 角色进行动态仿真，运行结果如图 7-2 所示，增加场景中的角色数量，图 7-3 和图 7-4 所示分别为 30 × 30 个和 100 × 100 个的 Girl 角色的仿真结果。从图中可以看出，该方法可以较好地实现角色群组场景的仿真。

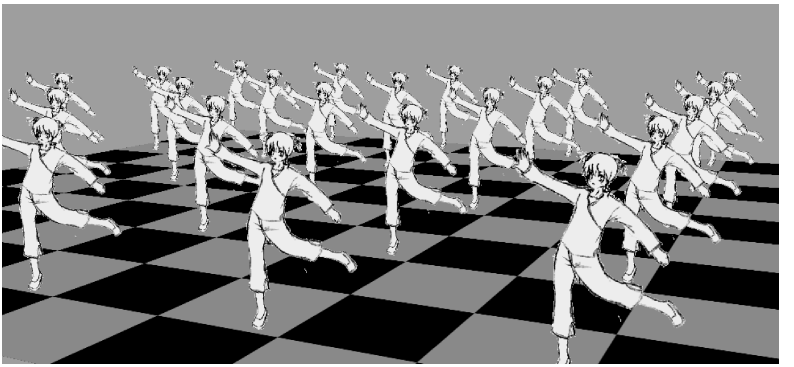


图 7-2　Girl 角色的仿真结果（5 × 5）

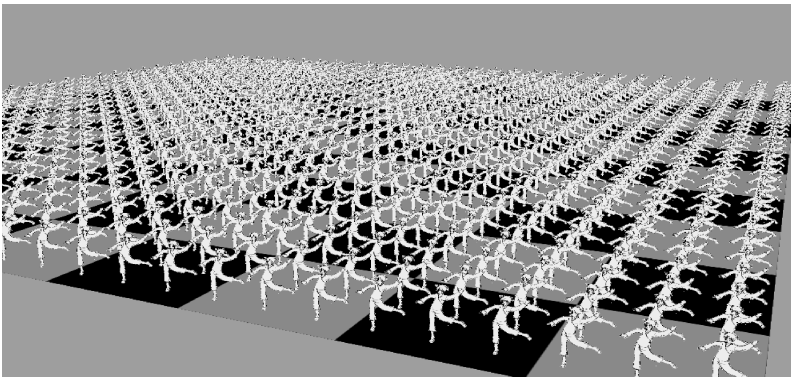


图 7-3　Girl 角色的仿真结果（30 × 30）

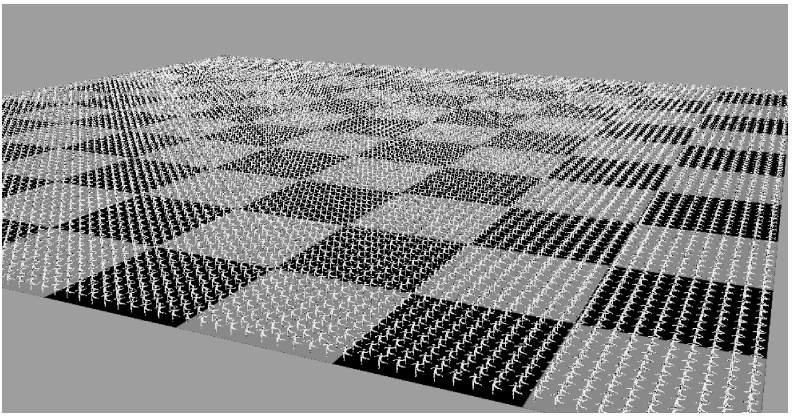


图 7-4　Girl 角色的仿真结果（100 × 100）

## 7.2 角色群组仿真速度与角色数量的关系

角色群组仿真中，对绘制速度影响最大的有两方面的因素——角色变形速度和角色绘制速度。本节的实验中，通过在一个场景中绘制大量保持运动状态的 Girl 角色，统计仿真过程中的角色变形时间和角色绘制时间，得出这两方面的运行时间受角色数量变化的影响规律。在实验中使用 10 个 Girl 角色进行复用，模拟场景中共有 10 种不同的运动角色，每次实验中连续执行 100 帧绘制，取各步骤执行时间的平均值，所获得的数据见表 7-1。

**表 7-1 Girl 角色仿真的各步骤执行时间** （单位：ms）

| 角色数量 | 角色投影大小 | 角色驱动 | 动态纹理生成 | 运动角色绘制 | 总时间 |
|---|---|---|---|---|---|
| 5×5 | 100×80 | 0.3654 | 3.721 | 3.556 | 7.6424 |
| 10×10 | 100×80 | 0.2896 | 3.700 | 9.278 | 13.2676 |
| 20×20 | 80×64 | 0.3137 | 3.892 | 27.14 | 31.3457 |
| 20×20 | 50×40 | 0.3252 | 4.212 | 12.80 | 17.3372 |
| 30×30 | 50×40 | 0.3257 | 3.779 | 28.45 | 32.5547 |
| 30×30 | 30×25 | 0.3642 | 3.965 | 12.14 | 16.4692 |
| 40×40 | 30×25 | 0.2824 | 3.815 | 21.53 | 25.6274 |
| 40×40 | 25×20 | 0.3350 | 3.956 | 15.23 | 19.5210 |
| 50×50 | 25×20 | 0.3521 | 4.133 | 23.87 | 28.3551 |
| 50×50 | 20×16 | 0.3285 | 3.831 | 16.38 | 20.5395 |
| 60×60 | 20×16 | 0.2785 | 3.831 | 23.69 | 27.7995 |
| 60×60 | 15×12 | 0.3261 | 3.914 | 14.56 | 18.8001 |
| 80×80 | 15×12 | 0.3123 | 4.013 | 25.75 | 30.0753 |
| 80×80 | 10×8 | 0.3761 | 4.103 | 13.52 | 17.9991 |
| 100×100 | 10×8 | 0.2972 | 3.876 | 20.66 | 24.8332 |
| 100×100 | 8×6 | 0.3328 | 3.949 | 13.39 | 17.6718 |
| 150×150 | 8×6 | 0.3776 | 3.860 | 28.93 | 33.1676 |
| 150×150 | 6×4 | 0.3105 | 3.833 | 15.84 | 19.9835 |

从表 7-1 中可以看出，使用本书提出的角色群组仿真方法，当场景中的角色数量在 150×150 规模时，每帧的角色绘制时间仍可保持在 30ms 以内，可以实现实时绘制。

表 7-1 中的数据还表明，场景中观察者的位置对于运动角色绘制时间的影响较大。观察者的位置，直接影响场景中的角色对屏幕的贡献率，当观察者离

运动角色距离更远时，角色将显得更小。因此，在场景中的运动角色数量增多时，为了能观察场景全景，必须使观察者位置拉远，造成每个角色的投影变小。通过合适地选择观察者位置，可以实现整个运动角色场景的实时绘制，场景中拥有不同运动角色数量时，要达到实时绘制，观察者位置与角色数量的关系如图 7-5 所示。

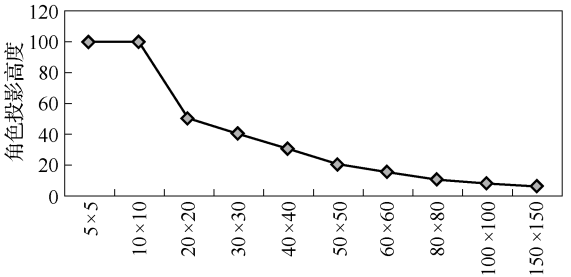



图 7-5　角色投影高度与角色数量的关系

## 7.3　角色群组仿真速度与角色种类数的关系

本书提出的角色群组仿真方法中，通过三种途径提高大规模角色的仿真速度：一是基于 Billboard 渲染；二是图形硬件加速；三是纹理复用。运动角色的种类数量只影响运动角色的动作驱动时间和动态纹理生成时间，本节仍使用上节的 Girl 角色，使场景中的角色总数量固定为 30 × 30，角色投影大小固定为 30 × 25，每一次实验连续记录 100 帧取平均值，所得执行时间记录见表 7-2。

**表 7-2　不同角色种类数下各步骤执行时间**　　（单位：ms）

| 不同角色种类数 | 角色动作驱动时间 | 动态纹理生成时间 |
|---|---|---|
| 5 | 0. 14665 | 3. 18 |
| 10 | 0. 3095 | 3. 72 |
| 15 | 0. 456 | 4. 278 |
| 20 | 0. 6162 | 4. 713 |
| 25 | 0. 8385 | 5. 355 |
| 30 | 0. 933 | 5. 72 |
| 35 | 1. 12875 | 6. 083 |
| 40 | 1. 3388 | 6. 556 |
| 50 | 1. 5835 | 7. 153 |
| 60 | 1. 8336 | 7. 908 |

根据表 7-2 中的数据，所绘制出的仿真各步骤执行时间与角色种类数的关系如图 7-6 所示。由图 7-6 中可以看出，角色驱动时间与角色种类数成正比，而动态纹理生成时间除环境设置和初始化时间外，也与角色种类数成线性增长关系。

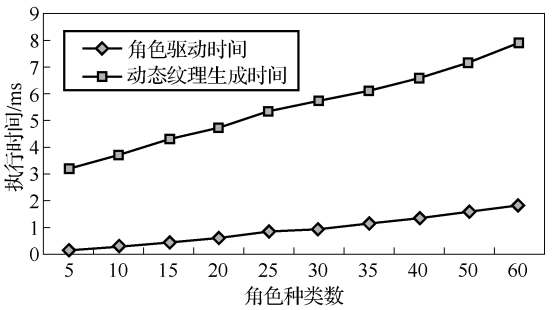



图 7-6　仿真各步骤运行时间与角色种类数的关系

# 第 8 章　群组仿真技术在森林场景中的应用

本书提出的结合 Billboard 图像渲染与动态纹理实现角色群组快速仿真的方法，还可以应用于其他具有大量近似模型的仿真中。在虚拟现实、战场仿真和数字娱乐等应用中，森林场景是一个重要的自然元素，但由于树木模型复杂、数量众多，而且对树木的受力动态效果更加难以仿真，使得实时渲染大规模的动态森林场景是一个具有挑战性的课题。

在动态森林场景仿真中，使用本书提出的二维角色变形技术实现树木图像在风力作用下的变形；使用动态纹理技术实时更新 Billboard 的纹理，并结合顶点偏移一起仿真树木模型在风力作用下的动态运动；利用图形硬件加速Billboard的顶点计算。通过这种方法可以实现大规模森林动态运动场景的实时仿真，对于几万株规模的动态森林场景，可以达到 30FPS 的仿真帧速率。

## 8.1　当前的森林场景仿真方法

多边形表示的树木模型拥有大量的多边形，从而使森林场景十分复杂，必须依靠加速手段才能实现快速可视化。Stephan[127] 给出了一个实时植被的渲染算法综述。Ferraro[128] 指出在森林场景中树木具有自相似特性。Anton[105] 结合图像简化与 LOD 方法表示树木模型，然后使用离散 LOD 方法快速渲染森林场景。但树木模型的结构并不利于面片的合并简化，LOD 层次模型的生成过程十分繁琐且效果并不好。

基于 Billboard 的图像渲染将多边形树木几何模型表达为平面图像，使用 Billboard 进行渲染[128]，Behrendt[129] 和 Deng[106] 均使用 Billboard 云方法实现若干树木的实时渲染。但传统的图像渲染只能绘制静止图像，无法实现动态森林场景仿真。

对于树木在风力下的动态效果仿真，Schaufler[101,102] 使用动态替代板技术，Max[104] 通过插值固定视点的纹理，柳有权[130] 运用硬件 shadow mapping 和 LOD 技术实现一棵树逼真的风力响应。2007 年宋成芳[131] 采用几何和图像混合均实现了小规模森林的动态仿真。但这些也均是小规模仿真，树木规模在几百株左右。

对于大规模的动态森林场景仿真，2004 年 Kurt[132] 通过偏移 Billboard 顶点位置模拟树木的动态效果，并利用图形硬件加速实现上万株树木的实时仿真，图

8-1所示为 Billboard 顶点偏移法示意图。

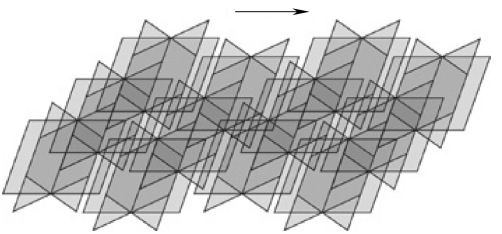


图 8-1 Billboard 顶点偏移法示意图

## 8.2 基于图像变形的单株树木变形方法

本节使用二维角色变形方法快速变形树木图像，根据树木图像标定变形区域，根据迎风受力面积计算各控制点的变形位移，从而实现树木图像变形。标定过程先使用边缘检测算法计算图像的树木有效区域，然后采用交互方式完成。图 8-2 所示为单株树木变形区域示意图。树木沿主干由底至上划为若干段（见图 8-2 左），最终获得该树木图像的受力区域（见图 8-2 右）。

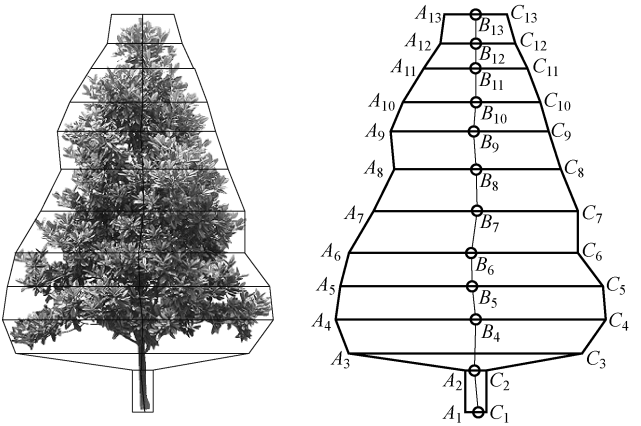



图 8-2 单株树木变形区域示意图

对于树木主干每一段，其迎风受力面积为所在区域的面积，如图中 $B_4B_5$段的迎风受力面积可用式（8-1）计算，其中 $\theta$ 是直线 $B_4B_5$的倾斜角。

$$Area_{45} = 1/2(|A_4C_4| + |A_5C_5|)|B_4B_5|\sin\theta \tag{8-1}$$

通过树木模型的单位面积的风作用力，可以计算该控制点上的受力大小。树木模型的单位面积受力变形系数可以用式（8-2）计算：

$$ratioF = \frac{k}{\sum_{i=1}^{n} Height_i(LenL_i + LenR_i)} \tag{8-2}$$

式中，$k$ 为调节系数；$Height_i$ 是第 $i$ 个控制点的控制高度；$LenL_i$ 和 $LenR_i$ 分别为其左右有效距离；$Height_i$（$LenL_i + LenR_i$）为各段的受力面积。

对于一株树木，每一段受风力影响的偏角为其上一段的偏角与自身的受力偏角之和，沿树根至树梢方向依次计算各段的倾斜角，并获得主干上的控制点（$B_1$，$B_2$，$B_3$，…，$B_n$）在运行时的位置。各控制点的新位置可由式（8-3）求得。

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} + Len_i \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \tag{8-3}$$

式中，$\theta$ 为当前控制点的倾斜角，其计算方法为该树木的单位面积受力变形系数与当前段面积之积，$\theta = ratioF \cdot Height$（$LenL + LenR$）。两侧边缘点的位置由中心控制点的位置及其夹角计算求得。图 8-3 所示为基于受力面积计算的树木变形结果与 Billboard 顶点偏移法变形结果的比较，其中图 8-3a 和图 8-3c 为顶点偏移法的变形结果，图 8-3b 和图 8-3d 是本章的图像变形方法的变形结果。

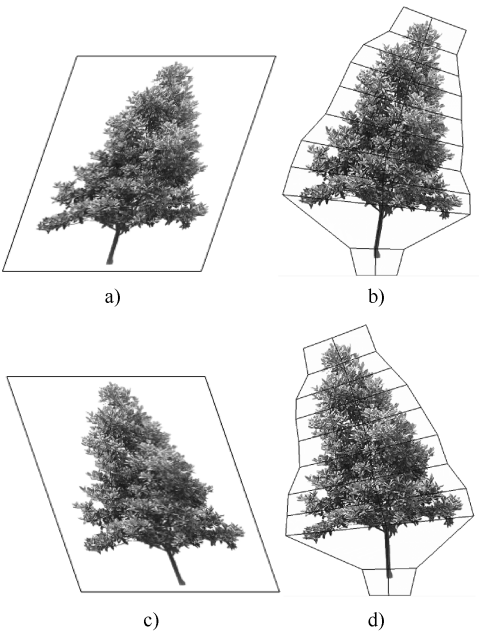



图 8-3　不同方法的树木变形结果比较

a）顶点偏移法结果　b）本章变形结果

c）顶点偏移法结果　d）本章变形结果

## 8.3 基于群组仿真技术的森林场景绘制方法

树木主要是由主干受风力弯曲变形，整个树木主干在同一时刻的受力仅仅限于单一方向，各个控制点的受力变形有较大相似性且相互影响。根据树木弯曲形变特性在水平面上的各向同性特点，可以将树木的受力形变分解为平行于视点方向和垂直于视点方向，对两个方向的形变分别处理，降低计算处理的复杂度。对垂直于视点方向的形变，使用 Billboard 顶点偏移的方法进行模拟。对平行于视点方向的形变，使用图像变形代替顶点偏移，提高树木真实感，并预计算树木变形结果，并将其组合成为复合纹理，提高仿真速度。

本节的仿真方法中，将风力由左向右分为 64 种风力条件，通过渲染到纹理技术，将单株树木在这些风力条件下的变形结果预先绘制为一个复合纹理。使用图 8-2 中的树木生成的复合纹理如图 8-4 所示。

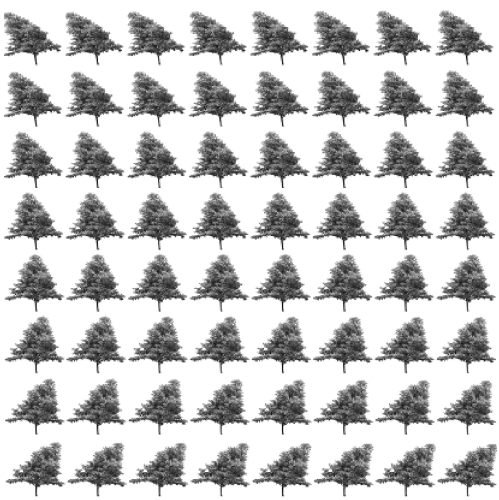


图 8-4 树木变形的复合纹理图

在动态森林场景仿真中，GPGPU 技术被用来加速计算 Billboard 的顶点坐标和复合纹理坐标。GPU 顶点程序的复合纹理坐标计算部分可以描述为

（1）根据所受的风力及其风向，计算该风力在横向方向上的分量；

（2）根据风力的横向分量，计算其对应的复合纹理序号；

（3）根据复合纹理的分块规则，计算所需纹理在复合纹理中的实际横序号和纵序号；

（4）根据顶点序号计算当前顶点的复合纹理坐标。

对于 Billboard 的复合纹理坐标计算，先获取该树在当前时刻所受的平行风力

分量，再据此计算该风力分量对应的变形结果在复合纹理中的编号，根据编号计算纹理坐标。风力分量对应的编号可用式（8-4）计算。

$$texNum = \begin{cases} 32(1 - force_x / MaxForce), left \\ 32(1 + force_x / MaxForce), right \end{cases} \tag{8-4}$$

Billboard 四个顶点的纹理坐标可分别用式（8-5）中的各个分量进行计算。

$$\begin{bmatrix} \left(\frac{x}{8}, \frac{y+1}{8}\right) & \left(\frac{x+1}{8}, \frac{y+1}{8}\right) \\ \left(\frac{x}{8}, \frac{y}{8}\right) & \left(\frac{x+1}{8}, \frac{y}{8}\right) \end{bmatrix} \tag{8-5}$$

式中，$x = texNum/8$，$y = texNum/8$ 是该 Billboard 的复合纹理编号。

## 8.4　基于群组仿真技术的动态森林仿真算法实现

本章基于群组仿真技术实现了大规模动态森林场景的仿真，动态森林场景的具体仿真过程可以描述为：

（1）计算场景中当前时间的风力分布，并根据树木位置获得所受的风力参数；

（2）针对每一株树木，根据风力的平行分量计算其复合纹理坐标；

（3）根据树木位置与视点位置，计算其替代 Billboard 的顶点坐标；

（4）根据风力垂直分量计算俯仰位移，调节顶点位置；

（5）使用动态纹理技术，实时更新纹理内容。

图 8-5 所示为一帧动态森林场景仿真的流程图。

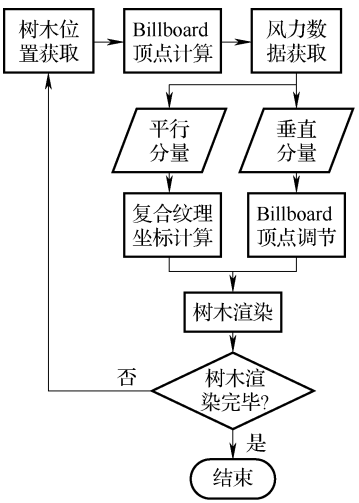



图 8-5　动态森林仿真流程图

## 8.5 大规模动态森林场景仿真关键代码

本书的大规模角色群组仿真方法，不仅可以应用于大量运动角色的仿真，对于具备运动特征的其他大规模物体同样可以应用该技术。在本书中给出了一个大规模动态森林场景的仿真场景。在大规模动态森林的仿真过程中，与群组动态角色仿真的不同之处有两点，一个是 Billboard 板会随视点的位置而旋转，另一个是需要在过程中生成复合纹理。其中 Billboard 板的顶点位置计算代码如下：

```
void ReCalBBVexPosCpu(float * eyePos31)
{
  // 通过顶点输入，计算 BB 板顶点的真实位置
  memcpy_s (EyePos3, 3* szfloat, eyePos31, 3* szfloat);
  //BB 板立点位置
  if(treeBBVexPos4x3 == NULL) treeBBVexPos4x3 =
       new float [12* treeNum +1];
  ASSERT (treeBBVexPos4x3 != NULL);
  float vectEye2BBx, vectEye2BBy;
  float vectE2BLen, vectRot;
  float localRotCrdx, localRotCrdy;
  // 普通方法绘制
  float * dataM = treeBBVexPos4x5;
  float * posBBCrd = treeBBVexPos4x3;
  for(int i =0; i < treeNum* 4; i ++)
   {
     vectEye2BBx = dataM [0] - EyePos3 [0];
     vectEye2BBy = EyePos3 [1] - dataM [1];
     vectE2BLen = sqrtf(vectEye2BBx* vectEye2BBx +
          vectEye2BBy* vectEye2BBy);
     // 旋转：四方框坐标坐标 ( +/ -0.5* width, 0)
     vectRot = BBWidth* (dataM [3] -0.5)/vectE2BLen;
     localRotCrdx = vectEye2BBy* vectRot;
     localRotCrdy = vectEye2BBx* vectRot;
     // 移动坐标至立点
     localRotCrdx += dataM [0];
     localRotCrdy += dataM [1];
```

```
        //localRotCrd [0, 1] 是实际立点的 XY 坐标
        * (posBBCrd ++) = localRotCrdx;
        * (posBBCrd ++) = localRotCrdy;
        * (posBBCrd ++) = BBHeight* dataM [4] + dataM [2];
        dataM + =5;
    }
}
```

## 8.6　实验结果与分析

基于本章的动态森林场景快速仿真的算法思想，现已经在 PC 机上实现了风作用下的大规模森林场景仿真，算法实现的硬件平台为 Intel 奔腾 4 3.0G CPU，1GB 内存，nVidia GeForce FX6600 128M 显卡，软件环境为 Windows XP 操作系统，编程环境为 Microsoft Visual Studio 2005，3D 开发环境为 OpenGL。

本节的实验主要包含如下几部分：

（1）验证通过群组仿真技术实现大规模动态森林场景快速仿真的可行性。通过动态森林场景的仿真结果，说明该方法的可行性和有效性。

（2）获得动态森林场景中树木数量对仿真速度的影响。通过分别统计不同数量树木的森林的各步骤执行时间，获得树木数量与仿真速度的关系。

### 8.6.1　基于群组仿真技术的动态森林绘制结果

本章的实验使用一幅树木图像实现动态森林场景的仿真，图 8-6 所示为 70 × 70株树木的动态森林场景的仿真结果，图 8-7 所示为 100 × 100 株树木的动态森林场景的仿真结果。从图中可以看出，该方法可以较好地实现大量树木的动态森林场景的仿真。

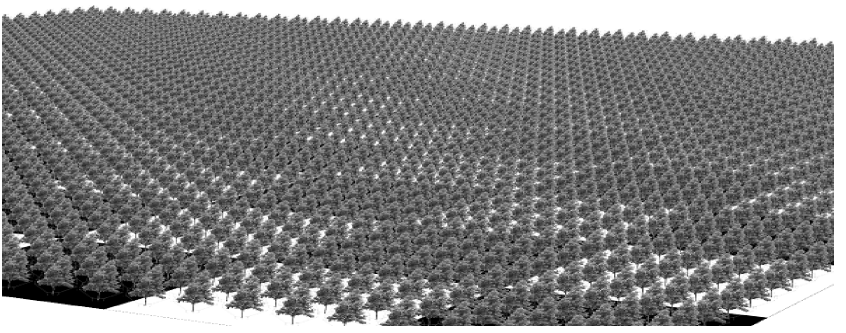


图 8-6　森林仿真结果（70 × 70）

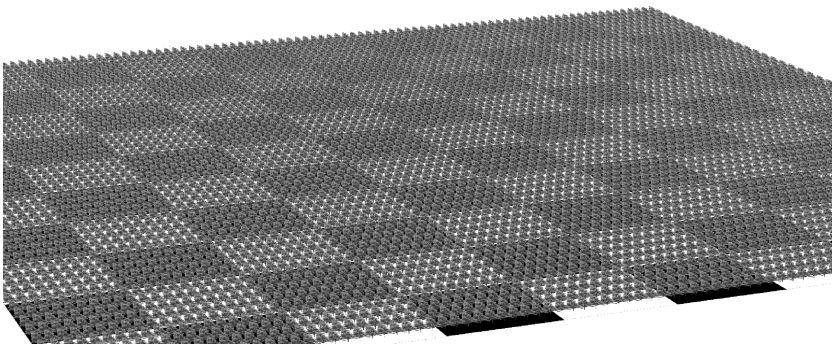


图 8-7 森林仿真结果（100 × 100）

## 8.6.2 动态森林仿真速度与树木数量的关系

动态森林场景仿真中，对仿真速度影响有显著影响的因素包括三方面：Billboard 顶点坐标计算、Billboard 纹理坐标计算和树木绘制时间。本实验中，通过绘制大量的保持运动状态的树木，统计仿真过程中的各步骤计算时间和树木绘制时间，得出执行时间随树木数量变化的规律。每次实验连续执行 100 帧绘制，取各步骤执行时间的平均值，所获得的数据见表 8-1。

**表 8-1 不同树木数量的仿真各步骤时间** （单位：ms）

| 树木数量 | 顶点坐标计算 | 纹理坐标计算 | 树木绘制 |
|---|---|---|---|
| 10 × 10 | 0. 02031 | 0. 2162 | 0. 5632 |
| 15 × 15 | 0. 04048 | 0. 284 | 0. 8214 |
| 20 × 20 | 0. 07877 | 0. 3309 | 1. 420 |
| 25 × 25 | 0. 1143 | 0. 7501 | 2. 069 |
| 30 × 30 | 0. 1529 | 0. 6023 | 3. 470 |
| 35 × 35 | 0. 198 | 1. 270 | 3. 984 |
| 40 × 40 | 0. 2548 | 1. 434 | 5. 421 |
| 45 × 45 | 0. 3661 | 2. 195 | 6. 737 |
| 50 × 50 | 0. 4234 | 1. 199 | 8. 255 |
| 60 × 60 | 0. 5713 | 1. 794 | 12. 04 |
| 70 × 70 | 0. 8547 | 2. 483 | 15. 76 |
| 80 × 80 | 1. 079 | 6. 072 | 21. 26 |
| 90 × 90 | 1. 445 | 4. 156 | 27. 43 |
| 100 × 100 | 1. 619 | 9. 462 | 33. 16 |

根据表 8-1 绘制的各个步骤的运行时间随树木数量变化的关系图如图 8-8 所示。

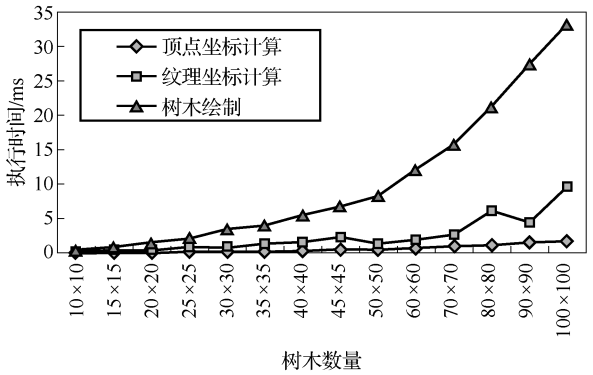



图 8-8　动态森林仿真时间与树木数量的关系

由图中可以看出，各步骤花费的时间均随树木数量增长，顶点坐标计算时间增长较缓，而树木绘制时间增长最快。使用本方法在本机器上所能实时仿真的树木数量为 1 万株左右。

# 第 9 章　总结与展望

## 9.1　本书的工作总结

大规模角色群组仿真是虚拟现实与计算机图形学领域的重要研究内容之一。在计算机生成场景中，运动角色是指运动状态下的人物、动物和植物等仿真实体，本书以运动角色为研究对象，对其具有真实感的大规模角色群组仿真技术展开研究，以支持有运动角色参与的计算机虚拟场景的仿真，如大型文艺编排、虚拟演习、虚拟训练等。本书以运动角色的仿真过程为主线，在大规模运动角色快速仿真的相关技术中进行了二维角色变形、三维角色变形、角色动作群组和角色快速绘制四部分内容的研究。针对现有技术中存在的不足，本书对上述关键技术进行创新性研究，主要研究工作如下：

**1. 研究了基于自适应网格的二维角色变形方法**

通过分析和研究目前的二维角色变形的算法和特点，提出了一种适用于二维角色变形的自适应变形网格，解决了二维角色变形中的不平滑失真和无法硬件加速的问题。在大规模角色群组仿真中，角色是最主要的对象，而变形计算是其主要计算内容。自适应变形网格使用曲线控制其形状变化以完成骨骼绕关节点的旋转动作；通过控制线的导数连续性保证二维角色的变形平滑性。通过面积保持降低变形失真；通过影响因素局部化和图形硬件加速保证计算的实时性。使用 GPGPU 技术加速自适应网格的顶点计算，降低了 CPU 的计算压力，以支持大量角色的实时变形计算。实验结果表明，该方法可以避免变形中的不平滑失真，可以借助 GPGPU 加速实现大量二维角色的快速变形。

**2. 研究了基于统一基础模型的三维角色变形方法**

通过分析和研究细节保持的三维模型变形特点，提出了一种使用规则网格直接构建的统一基础模型，解决了三维模型变形过程中基础模型建立复杂、无法进行硬件加速的问题。在三维模型变形过程中，统一基础模型通过直接构建与原模型相关性很小的规则网格，避免基础模型的生成计算；通过基于表面法向量的径向计算进行模型细节的分离和合成，实现细节保持的变形；使用统一基础模型，可以对于同类三维模型使用统一的变形算法；通过叠加阻尼振荡曲线，模拟变形物体受挤压产生的表面皱褶，在实现细节保持的同时实现模型表面积保持，提高变形真实感；通过规则网格的易处理特性和图形硬件加速，保证算法的实时性。

实验结果表明，该方法可以实现三维模型的快速变形。

**3. 研究了基于动作迁移的角色驱动方法**

通过分析和研究目前的动作数据建立和角色自动驱动的方法，提出了一种基于动作迁移的角色驱动方法，解决了运动角色仿真中动作配置繁琐、动作数据共享性差的问题。该方法将现有动作数据转换为统一的基于关节点旋转角的动作数据，方便的应用到角色驱动中；使用一种叠加无配对的关节点旋转角的方法，实现不同拓扑结构的两个角色间的动作迁移；通过 2D→3D→2D 的转换方法，对齐角色姿态，实现两个二维角色之间的动作迁移，解决了以往动作迁移无法应用于二维角色目标的问题。实验结果表明，该驱动方法可以自动驱动虚拟角色，并可以产生与源角色完全相同的动作序列。

**4. 研究了基于动态纹理的角色绘制与仿真方法**

通过分析目前的图形硬件特性，提出了一种适用于大规模角色绘制的动态纹理技术，结合 Billboard 实现大量角色的快速绘制。该方法借助于离屏渲染技术，在运行时将一个渲染过程渲染为内容实时改变的动态纹理，在显存中直接应用于角色绘制；使用 Billboard 作为三维场景中运动角色载体，通过纹理复用快速绘制大量活动角色，隐性减少变形计算和模型数量；使用 GPGPU 技术进行 Billboard 顶点位置的加速计算，保证大规模运动角色的实时可视化。实验结果表明，该方法可以很大程度上提高实时绘制的角色规模。

**5. 研究了角色变形和角色绘制过程中的图形硬件加速方法**

提出了应用于角色变形和角色绘制的图形硬件加速方法。利用图形硬件的可编程性和执行并行性的特点，在运动角色仿真过程中，通过改进变形方法和绘制方法，采用 GPGPU 图形硬件加速提高仿真速度。改进算法并实现了两方面的硬件加速：一方面是在二维角色变形中使用 GPGPU 技术实现了自适应网格顶点坐标计算；另一方面是在角色快速绘制中借助于离屏渲染技术生成动态纹理，并使用 GPU 加速绘制技术实现 Billboard 顶点坐标、纹理坐标计算。实验数据表明，这些加速方法对运动角色的仿真速度具有很大提高。

## 9.2　进一步研究与展望

本书在前人的工作基础之上，针对若干关键技术进行了探索性的研究，取得了一定的成果。但是，还有许多问题值得进一步的研究和探讨，进一步的工作主要围绕以下几个方面展开：

**1. 进一步完善当前的角色变形方法**

本书中研究了角色的变形方法，进一步工作将研究二/三维角色变形的合理性约束方法，并研究使用更少的交互实现角色变形的方法。对于三维角色变形将

进行基于复杂控制曲线的变形技术研究，以及基于统一基础模型的三维模型变形的硬件加速实现方法。对于角色驱动方面，将研究复杂动作序列的驱动方法。

**2. 二/三维结合的角色仿真技术研究**

本书分别实现了二维运动角色的仿真和三维运动角色的仿真，二维/三维角色各有其优缺点，分别适用于不同的场合。但这种场合的划分，并没有很明显的界限，研究二/三维结合的角色仿真技术，可以在同一个场景中，根据机器配置、屏幕贡献或人为设置等进行自动切换，在近处使用三维角色，在远处使用二维角色，实现更佳的仿真效果。

**3. 具有多角色交互特性的角色仿真技术研究**

在本书的角色仿真方法中，各个角色之间的交互并没有被详细考虑，进一步的工作将致力于研究含有多角色交互的大规模运动角色场景绘制方法，以及有真实人物参与交互的运动角色仿真。虚拟环境中的角色与现实中的人形成回路控制，将是一个前景广泛的研究方向。

**4. 大规模角色群组的群体行为建模研究**

对于虚拟场景中的群体行为，仅仅拥有大规模角色群组仿真技术是不够的，对于群体行为的建模，如个体之间的影响和交互等，对于社会管理、人群模拟、紧急事件演习、虚拟文艺表演等都是重要的研究方向。

**5. 网络环境下的角色群组仿真技术研究**

目前还没有一种有效的基于分布式的大规模角色群组仿真方法。分布式的仿真方法除能进行更流畅的计算和更大规模的仿真外，更重要的是可以由各地的操作人员进行联合仿真，人机合作和基于虚拟环境的远程协作将是虚拟现实发展的方向。

# 参 考 文 献


[1] 汪成为. 人类认识世界的帮手——虚拟现实 [M]. 北京：清华大学出版社，2000.

[2] 张茂军. 虚拟现实系统 [M]. 北京，科学出版社，2002.

[3] Bouvier E，Cohen E，Najman. L. From crowd simulation to airbag deployment：particle systems，a new paradigm of simulation [J]. Journal of Electronic Imaging 6 (1)，1997 (1)：94-107.

[4] Franco Tecchia，Yiorgos Chrysanthou. Real-Time Rendering of Densely Populated Urban Environments [C]. Proceedings of the Eurographics Workshop on Rendering Techniques 2000：83-88.

[5] S Musse，D Thalmann. A Model of Human Crowd Behavior：Group Inter-Relationship and Collision Detection Analysis [C]. Proc. Workshop of Computer Animation and Simulation of Eurographics. 97，Budapest，Hungary. 1997：39-51.

[6] Musse S R，Thalmann D. A Behavioral Model for Real Time Simulation of Virtual Human Crowds [J]. IEEE Transactions on Visualization and Computer Graphics 2001，7 (2)：152-164.

[7] 王兆其. 虚拟人合成研究综述 [J]. 中国科学院研究生院学报，2000 (2)：89-98.

[8] Badler N I，Bindinganavale R，Granieri J P.，et al. Posture Interpolation with Cpllision Avoidance [C]. In Proceedings of Computer Animation 1994：13-20.

[9] Capin T K，Pandiz I S，Noser H. et al. Virtual Human Representation and Communication in VLNET [J]. IEEE Computer Graphics and Applications，1997，17 (2)：42-53.

[10] Hodgins J K，Wooten W L，Brogan D C. et al. Animating Human Atheletics [C]. In Proceedings of SIGGRAPH 95，1995：71-78.

[11] Moeslund TB，Hilton A，Kruger V. A survey of advances in vision-based human motion capture and analysis [J]. Computer Vision and Image Understanding，2006，104：90-126.

[12] Beier T，Neely S. Feature-Based image morphing [J]. Computer Graphics，1992，26 (2)：35-42.

[13] Sederberg T W，Parry P. Free-form deformation of solid geometric models [C]. Computer Graphics (Siggraph'86)，1986，20 (4)：151-160.

[14] Stalpers M G J R，Overveld C W A M. Deforming geometric models based on a polygonal skeleton mesh [J]. Journal of Graphics Tools archive，1997，2 (3)：1-14.

[15] Takeo Igarashi，Tomer Moscovich，John F，et al. As-Rigid-As-Possible Shape Manipulation [J]. ACM Transactions on Computer Graphics，2005，24 (3)：1134-1141.

[16] Andrew Nealen，Olga Sorkine，Marc Alexa，et al. A sketch-based interface for detail-preserving mesh editing [C]. ACM SIGGRAPH 2007，San Diego，California，courses，August，2007：05-09.

[17] Tan Kim Heok，Daut Daman，A Review on Level of Detail [C]，Proceedings of the International Conference on Computer Graphics，Imaging and Visualization，2004：70-75.

[18] Kobbelt L，Botsch M. A survey of point-based techniques in computer graphics [C]. Computer&Graphics，2004，28 (6)：801-814.

[19] Heung-Yeung Shum, Sing Bing Kang. A review of image-based rendering techniques [C]. Proceedings of IEEE/SPIE Visual communications and Image Processing. Perth, Australia, 2000: 2-13.

[20] Cg shader website. http: //www. cgshader. org [OL].

[21] gpgpu website. http: //www. gpgpu. org [OL].

[22] G. Maestri. Digital Character Animation 2 [M], Vol 1. New Rider, Indianapolis, Amazon. com, 1999.

[23] H Ko, N Badler. Animating human locomotion in real-time using inverse dynamics, balance and comfort control [J]. IEEE Computer Graphics and Applications, 1996, 16 (2): 50-59.

[24] Magnenat-Thalmann N, Laperriere R, Thalmann D. Joint-Dependent Local Deformations for Hand Animation and Object Grasping [C]. Proc. Graphics Interface, 1988: 26-33.

[25] Magnenat-Thalmann N, Thalmann D. Human Body Deformations using Joint-Dependent Local Operators and Finite Element Theory [A]. Making Them Move: Mechanics, Control and Animation of Articulated Figures San Mateo, CA: Morgan Kaufmann, 1991: 243-262.

[26] Nirut Naksuk, C. S. George Lee, Shirley Rietdyk. Whole-Body Human-to-Humanoid Motion Transfer [C]. Proceedings of 2005 5th IEEE-RAS International Conference on Humanoid Robots,2005: 104-109.

[27] Yoshimoto H, Date N, Yonemoto S. Vision-based real-time motion capture system using multiple cameras [C], Proceedings of MFI2003 (IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems), 2003: 247-251.

[28] Wu J C, Popovic Z. Realistic modeling of bird flight animations [C]. ACM Transactions on Graphics (SIGGRAPH 2003), July 2003, 22: 888-895.

[29] Kehl R. Markerless Motion Capture of Complex Human Movements from Multiple Views [D]. A dissertation of the Swiss Federal Institute Of Technology Zurich, 2005.

[30] Li H J, Lin S X, Zhang Y D. A survey of video based human motion capture [J]. Journal of Computer Aided Design & Computer Graphics, 2006, 18: 1645-1651.

[31] Wan C K, Yuan B Z, Sun Y D, et al. Human Body Motion Capture via 3D Graph-cuts [C]. Proceedings of The 8th International Conference on Signal Processing, 2006, 12: 16-20.

[32] Pei Y R, Zha H B. Transferring of Speech Movements from Video to 3D Face Space [C]. IEEE Trans Vis Comput Graph 2007. 13 (1): 58-69.

[33] Ana-MariaCretu, Dipl Eng. 3D Object Modeling-Issues and Techniques [R]. Preliminary Report, SITE, University of Ottawa, 2003. 10.

[34] Gordon Collins, Adrian Hilton. Models for Character Animation [R/OL]. Course Note of Centre for Vision, Speech and Signal Processing University of Surrey, 2001, Guildford GU25XH, UK. http: //www. ee. surrey. ac. uk/Research/VSSP/3DVision.

[35] Casale M S, Stanton E L. An overview of analytic solid modeling [J]. IEEE Computer Graphics&Applications, 1985, 5 (2): 45-56.

[36] Gibson S F, Mirtich B. A Survey of Deformable Modeling in Computer Graphics [R]. Technical report TR-97-19, Mitsubishi Electric Research Laboratories, 1997.

[37] Barr A H. Global and local deformations of solod primitives [J]. Computer Graphics, 1984, 18 (3): 21-23.

[38] Lewis J, Cordner M, Fong N. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation [C]. Proceedings of the ACM SIGGRAPH Conference on Computer Graphics, 2000: 165-172.

[39] Wolberg, W. Digital Image Warping [M]. Los Alamitos, CA: IEEE Computer Society Press, 1990.

[40] Alexa M, Cohen-Or D, Levin D. As-rigid-as-possible shape interpolation [C]. ACM Siggraph, 2000: 157-164.

[41] Peter-Pike J Sloan, Charles F Rose III, Michael F Cohen. Shape by example [C]. proceedings of 2001 symposium on interactive 3d graphics 2001: 135-143.

[42] 偶春生, 张佑生, 高月芳. 基于区域边界的图像变形算法 [J]. 系统仿真学报, 2003 (7): 976-979.

[43] Martin K, Niloy J M, Helmut P. Geometric modeling in shape space [C]. proceeding of Acm Siggraph conferences 2007: 64-1-8.

[44] He Guo, Xinyuan Fu, Feng Chen, et al. As-rigid-as-possible shape deformation and interpolation [J]. Journal of Visual Communication and Image Representation, 2008, 19 (4): 245-255.

[45] Coquillart S. Extended free-form deformation: a sculpturing tool for 3D geometric modeling [C]. Computer Graphics (Siggraph90), 1990, 24 (4): 187-196.

[46] Coquillart S, Jancene P. Animated free-form deformation: an interactive animation technique [C]. Computer Graphics (Siggraph91), 1991, 25 (4): 23-26.

[47] Kalra P, M angiliA, Thalmann D, et al. Simulation of facial actions based on rational free-form deformation [C]. Computer Graphics Forum (Eurographics92), 1992, 2 (3): 59-69.

[48] Lamousin H J, Waggenspack W N. NURBS-based free-form deformation [J]. IEEE Transactions on Computer Graphics and Applications, 1994, 14 (6): 59-65.

[49] Celniker G, Gossard D. Deformable curve and surface finite-elements for free-form shape design [C]. ACM Siggraph, 1991: 257-266.

[50] MacCracken R, Kenneth I J. Free-form deformations with lattices of arbitrary topology [C]. Computer Graphics, Proceedings of Siggraph 1996, 181-188.

[51] Singh K, Fiume E. Wires: a geometric deformation technique [C]. proceedings of ACM siggraph 1998: 405-414.

[52] Clint Chua, Ulrich Neumann. Hardware-accelerated free-form deformation [C]. Proceedings of the ACM Siggraph/Eurographics workshop on Graphics hardware, August 21-22, 2000: 33-39.

[53] Milliron T, Jensen R, Barzel R, et al. A Framework for Geometric Warps and Deformations [J]. ACM Trans. Graphics, 2002, 21 (1): 20-51.

[54] Draper G, Egbert P. A gestural interface to freeform deformation [C]. In Proceedings of Graphics Interface 2003: 113-120.

[55] Faloutsos P, Vandepanne M, Terzopou-Los D. Dynamic freeform deformations for animation

synthesis [J]. IEEE Transactions on Visualization and Computer Graphics, 1997, 3 (3): 201-214.

[56] Lewis J P Cordner M, N. Fong (Matt Cordner, Nickson Fong). Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation [C]. Proceedings of the ACM Siggraph Conference on Computer Graphics, 2000: 165-172.

[57] Capell S, Green S, Curless B, et al. Interactive Skeleton-Driven Dynamic Deformations [C]. ACM Transactions on Graphics, 2002, 21 (3): 586-593.

[58] Paul G K, Doug L J, Dinesh K P. Eigenskin: real time large deformation character skinning in hardware [C]. proceeding of the Acm siggraph symposium on Computer Animation, 2002: 153-159.

[59] Mohr A, Gleicer M. Building efficient, accurate character skins from examples [C]. Acm trans, graph, 2003: 562-568.

[60] Sumner R W, Zwicker M, Gotsman C, et al. Mesh-based inverse kinematics [C]. ACM Trans. Graphics (Proc. Siggraph2005) 2005, 24 (3): 488-495.

[61] HanBing Yan, ShiMin Hu, Ralph Martin. Skeleton-Based Shape Deformation Using Simplex Transformations [C]. Proceedings of Advances in Computer Graphics. CGI 2006 (Lecture Notes in Computer Science, 2006, 4035): 66-77.

[62] Kevin G D, Sumner R W, Popovic J. Inverse Kinematics for Reduced Deformable Models [C]. ACM Transactions on Graphics-Proceedings of ACM SIGGRAPH 2006: 1174-1179.

[63] Jyh-Ming Lien, John Keyser, Nancy M Amato. Simultaneous Shape Decomposition and Skeletonization [C]. Proc. ACM Solid and Physical Modeling Symp. (SPM), Cardiff, UK, 2006 (6): 219-228.

[64] Chang Y K, Rockwood A P. A generalized de Casteljau approach to 3D free-form deformation [J]. Computer Graphics, 1994, 28 (3): 257-260.

[65] Lazarus F, Coquillart S, Jancene P. Axial deformations: an intuitive deformation technique [J]. Computer Aided Design, 1994, 26 (8): 607-612.

[66] 金小刚, 彭群生. 基于弧长不变的曲线变形 [J], 软件学报, 1997 (8): 49-55.

[67] Peng Qunsheng, Jin Xiaogang, Feng Jieqing. Arc-length-based axial deformation and length preserving deformation [C]. Proc. of Computer Animation'97, Geneva, IEEE Computer Society, 1997, 86-92.

[68] Xiaosong Yang, Somasekharan Arun, Zhang Jian J. Curve skeleton skinning for human and creature characters [J]. Computer Animation and Virtual Worlds, 2006-6, 17 (3-4): 281-292.

[69] Cornea Nicu D. Patrick Min, Silver D. Curve-Skeleton Properties, Applications and Algorithms [C]. IEEE Transactions on Visualization and Computer Graphics, v 13, n 3, May/June, 2007: 530-548.

[70] Shi X H, Zhou K, Tong Y Y, et al. Mesh Puppetry: Cascading Optimization of Mesh Deformation with Inverse Kinematics [C]. siggraph2007, ACM Transactions on Graphics (TOG) 2007, 26 (3) 81-110.

[71] Capell S, Green S, Curl Ess B, et al. Interactive skeleton-driven dynamic deformations [C]. Proceedings of SIGGRAPH . San Antonio, ACM, 2002: 586-593.

[72] 郁佳荣, 石教英, 周永霞 . 基于物理的实时人体动画 [J]. 浙江大学学报: 工学版, 2006, 40 (12): 2079-2082.

[73] Scott Schaefer, Travis McPhail, Joe Warren. Image Deformation Using Moving Least Squares [C]. ACM Siggraph 2006, Session: Image processing: 533-540.

[74] Yanlin Weng, Weiwei Xu, Yanchen Wu, et al. 2D Shape Deformation Using Nonlinear Least Squares Optimization [C]. The Visual Computer: International Journal of Computer Graphics, 2006 (Sep), Volume 22: 653 - 660.

[75] Yasuhiro Yoshioka, Hiroshi Masuda, Yoshiyuki Furukawa. A Constrained Least Squares Approach to Interactive Mesh Deformation [C]. IEEE International Conference on Shape Modeling and Applications, 2006: 153-162.

[76] O Sorkine, Y Lipman, D Cohen2Or, et al. Laplacian surface editing [C], Proc of the Eurographics PACM SIGGRAPH Symposium on Geometry Processing. New York: ACM Press, 2004: 179-188.

[77] Zhou Yu Y, Xu K, Shi D, et al. Mesh editing with poisson-based gradient field manipulation [C]. ACM Trans. Graph, 2004, 23 (3): 644-651.

[78] Alexa M. Differential coordinates for local mesh morphing and deformation [J]. The Visual Computer 2003, 19 (2): 105-114.

[79] Y Lipman, O Sorkine, D Cohen-Or, et al. Differential coordinates for interactive mesh editing [C]. In : Proc of Shape Modeling International, Washington : IEEE Computer Society Press, 2004: 181-190.

[80] Marc Alexa, Mesh editing based on discrete Laplace and Poisson models [R]. ACM SIGGRAPH 2006 Courses, July 30-August 03, 2006, Boston, Massachusetts.

[81] Wu J Z, Liu X H, Wu E H. Mesh Deformation under Skeleton-based Detail-preservation [C]. 10th IEEE International Conference on Computer-Aided Design and Computer Graphics, 2007: 453-456.

[82] Kobbelt L, Vorsatz J, H 2P Seidel. Multi-resolution hierarchies on unstructured triangle mesh [J]. Computer Geometry, 1999, 14 (1): 5-24.

[83] Botsch M, Kobbelt L. Multiresolution surface representation based on displacement volumes [J]. Computer Graphics Forum, 2003, 22 (3): 483-491.

[84] M G Choi, H-S Ko. Modal warping : Realtime simulation of large rotational deformation and manipulation [J] . IEEE Trans. on Visualization and Computer Graphics, 2005, 11 (1):91-101.

[85] Huang J, Shi X, Liu X, et al. Subspace Gradient domain mesh deformation [C]. Acm trans graph, 2006: 1126-1141.

[86] Oscar K C A, Fu H B, Tai C L, et al. Handle-Aware Isolines for Scalable Shape Editing [C]. proceeding of Acm Siggraph conferences, 2007, 26 (3): 83-1-83-10.

[87] 吴恩华, 刘学惠 . 虚拟现实与真实感图形生成 [J]. 中国图像图形学报, 1997, 2 (4):

205-212.

[88] 何晖光，田捷，张晓鹏，等．网格模型化简综述［J］．软件学报，2002，13（12）：2215-2224.

[89] 潘志庚，马小虎，石教英．多细节层次模型自动生成技术综述［J］．中国图形图像学报，1998，3（9）：754-759.

[90] Chadwick J E，Haumann D R，Parent R E. Layered construction for deformable animated characters［C］. Proceedings of Siggraph. Boston：ACM，1989：243-252.

[91] Hoppe H，DeRose T，Duchamp，T，et al. Mesh optimization［C/OL］. Computer Graphics Proceedings，1993：19-26. http：//citeseer. nj. nec. com/ hoppe93mesh. html.

[92] Hugues Hoppe. Progressive Meshes［C］. Proceedings of the ACM Siggraph Conference on Computer Graphics，1996：99-108.

[93] Cignoni P，Montani C，Scopigno R. A comparison of mesh simplification algorithm［R］. Teaching Report in Computer & Graphics，1998.

[94] Ogren A. Continuous level of detail in real-time terrain rendering［D］. Sweden Umea University，2000.

[95] 张昌明，张虹．一种基于边折叠的 LOD 自动生成算法［J］．计算机工程与设计，2005，26（11）：3109-3111.

[96] Sing Bing Kang，A Survey of Image-based Rendering Techniques［R/OL］. Cambridge Research Laboratory，August 1997，http：//www. hpl. hp. com/techreports/Compaq-DEC/.

[97] Cha Zhang and Tsuhan Chen，A Survey on Image-Based Rendering［R/OL］. Technical Report AMP 03-03 June 2003，Advanced Multimedia Processing Lab，http：//amp. ece. cmu. edu/Publication/.

[98] Yoshinori D，Kaneda K，Yamashita H，et al. A simple，efficient method for realistic animation of clouds［C］. Proceedings of ACM SIGGRAPH 2000. New Orleans：Louisiana，2000：19-28.

[99] Harris M J. Real-time cloud simulation and rendering［D］. Carolina：University of North Carolina，2003.

[100] Roditakis A. Modeling and visualization of clouds from real world data［R］. Federal Institute of Technology Zurich，2004.

[101] Marinov M，Botsch M，Kobbelt L. GPU-based multiresolution deformation using approximate normal reconstruction［J］. Journal of Graphics Tools，2007，12（1）：27-46.

[102] Schaufler G. Dynamically generated Impostors［C］. GI Workshop Modeling-Virtual Worlds-Distributed Graphics，1995：129-135.

[103] Schaufler G. Nailboards：A Rendering Primitive for Image Caching in Dynamic Scenes［C］. Eurographics Rendering Workshop，1997：151-162.

[104] Max N，Ohsaki K. Rendering Trees from precomputed z-buffer views［C］. In ：Eurographics Workshop on Rendering，Dublin，Ireland，1995：45 - 54.

[105] Anton F，Eike U，Stephan M. Extreme Model Simplification for Forest Rendering［D/OL］. VRV is Technical Report：Extreme Model，2004，http：//www. vrvis. at/vr/billboardclouds/.

[106] Deng Q Q，Zhang X P，Lei X D，et al. Fast Forest Visualization on Hierarchical Images and

Visibility [C]. Proceedings of The 2nd International Conference of E-Learning and Games-Edutainment, 2007: 44-55.

[107] Owens J D, Luebkk D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware [C]. Proceedings of the Eurographics, 2005: 21-51.

[108] 吴恩华, 柳有权. 基于图形处理器 (GPU) 的通用计算 [J]. 计算机辅助设计与图形学学报: 2004, 16 (5): 601-612.

[109] Kruger Jens, Westermann Rudiger. Linear algebra operators for GPU implementation of numerical algorithms [J]. ACM Transactions on Graphics, 2003, 22 (3): 908-916.

[110] Joao Luiz Dih. Comba, Dietrich Carlos A, Pagot Christian A, et al. Computation on GPUs: From a programmable pipeline to an efficient stream processor [J]. Revista de InformaticaTearicae Aplicada, 2003 (2): 41-70.

[111] Macedonia Michael. The GPU enters computing's mainstream [J]. Computer, 2003, 36 (10): 106-108.

[112] Li Wei, Wei Xiaoming, Kaufman Arie. Implementing lattice Boltzmann computation on graphics hardware [J]. The Visual Computer, 2003, 9 : 44-456.

[113] Kim Theodore, Lin Ming C. Visual simulation of ice crystal growth [C]. In : Proceedings of SIGGRAPH/ Eurographics Symposium on Computer Animation, San Diego, 2003: 86-97.

[114] 柳有权. 基于物理的计算机动画及其加速技术的研究 [D]. 北京: 中国科学院, 2005.

[115] Nvidia gpu site. http: //www. nvidia. com/object/gpu. html [OL].

[116] nvidia developer site. http: //developer. nvidia. com/ [OL].

[117] ATI developer site. http: //www. ati. com/developer/ [OL].

[118] microsoft directx site. http: //www. microsoft. com/directx/ [OL].

[119] opengl website. http: //www. opengl. org [OL].

[120] Wynn C. OpenGL Render-to-Texture [OL]. nVIDIA Corporation, white paper Edition. http://developer. nvidia. com/object/gdc_oglrtt. html, 2002.

[121] Zhou K, Jin Huang, Snyder J, et al. Large Mesh Deformation Using the Volumetric Graph Laplacian [C]. ACM Transactions on Graphics, 2005, 24 (3): 496-503.

[122] Brett Allen, brian Curless, Zoran Popovic. Articulated body deformation from range scan data [C]. ACM transactions on graphics, proceedings of siggraph 2002: 612-619.

[123] Cheung G K M, Baker S, Hodgins J, et al. Markerless human motion transfer [C]. Proceedings of 2nd International Symposium on 3DPVT (3D Data Processing, Visualization and Transmission), 2004: 373-378.

[124] Joshua L. Fast View-Dependent Level-of-Detail Rendering using Cached Geometry [C]. Proceedings of IEEE Visualization, 2002: 259-266.

[125] Endo T, Katayama A, Tamura H, et al. Image-based walk-through system for large-scale scenes [C]. Proceedings of VSMM98: 4th International Conference on Virtual Systems and Multimedia, 1998, 1: 269-274.

[126] Decoret X, Durand F, Sillion F X, et al. Billboard clouds for extreme model simplification [C]. ACM Transaction Graphics, 2003, 22 (3): 689-696.

[127] Stephan Mantler, Tobler Robert F, Fuhrmann Anton L. The state of the art in realtime rendering of vegetation [R]. Technical report, VRVis Research Center for Virtual Reality and Visualization, Vienna, Austria, 2003.

[128] Ferraro P, Godin C, Prusinkiewicz P. Toward a quantification of self-similarity in plants [C]. Fractals, 2005, 13: 91-109.

[129] Behrendt S, Colditz C, Franzke O, et al. Realistic Real-time Rendering of Landscapes Using Billboard Clouds [C]. Proceedings of Eurographics 2005, 24 (3): 507-516.

[130] 柳有权, 王文成, 吴恩华. 快速真实地生成树的自然摇曳 [J]. 计算机学报: 2005 (328): 1185-1191.

[131] 宋成芳, 谈奇峰, 张龙, 等. 风场作用下的动态森林场景的实时仿真 [J]. 计算机辅助设计与图形学学报, 2007, 19 (3): 323-328.

[132] Kurt Pelzer. GPU Gems [M]. nVidia. 2004: 107-121.

ISBN 978-7-111-33883-3
策划编辑：牛新国
封面设计：路恩中

地址：北京市百万庄大街22号　　邮政编码：100037
电话服务
社服务中心：(010)88361066
销售一部：(010)68326294
销售二部：(010)88379649
读者购书热线：(010)88379203
网络服务
门户网：http://www.cmpbook.com
教材网：http://www.cmpedu.com
封面无防伪标均为盗版

上架指导：计算机／计算机图形学

ISBN 978-7-111-33883-3

定价：39.80元