Generating, Animating, and Rendering Varied Individuals for Real-Time Crowds

THÈSE N^{O} 4405 (2009)

PRÉSENTÉE LE 22 MAI 2009

À LA FACULTE INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE RÉALITÉ VIRTUELLE

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Jonathan MAIM

acceptée sur proposition du jury:

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2009

CONTENTS

1 Introduction **9**

1.1 Motivations and Contributions . . . 10

1.1.1 *YaQ* Architecture . . . 10

1.1.2 Rendering Levels of Detail . . . 11

1.1.3 Crowd Variety . . . 11

1.2 Summary of Chapters . . . 13

2 Related Work **15**

2.1 Real-time Crowd Visualization . . . 15

2.1.1 Point-based Approach . . . 16

2.1.2 Image-based Approach . . . 17

2.2 Crowd Appearance Variety . . . 19

2.2.1 Color Variety . . . 19

2.2.2 Shape Variety . . . 20

2.3 Crowd Motion Planning and Navigation . . . 22

2.4 Crowd Architectures . . . 24

3 Overview **27**

3.1 YaQ Architecture . . . 27

3.2 Virtual Human Representations . . . 28

3.2.1 Human Template . . . 29

3.2.2 Deformable Mesh . . . 30

3.2.3 Rigid Meshes . . . 32

3.2.4 Impostor 32
3.3 *YaQ* Data 35
3.3.1 Human Template Data 35
3.3.2 Database Management 36

4 Appearance Variety **39**
4.1 Appearance Variety at Three Levels 40
4.2 Previous Work on Color Variety 41
4.2.1 Principles 42
4.2.2 HSB Color Spaces 42
4.3 Segmentation Maps 45
4.3.1 Principles 45
4.3.2 Color Computation and Storage 48
4.4 Conclusion 50

5 Shape Variety **53**
5.1 Accessories 54
5.1.1 Simple Accessories 54
5.1.2 Complex Accessories 55
5.2 Levels of Detail for Accessories 56
5.2.1 Rigid Meshes 57
5.2.2 Impostors 58
5.3 Accessories Implementation 61
5.3.1 Loading and Initialization 61
5.3.2 Rendering 62
5.3.3 Empty Accessories 64
5.3.4 Color Variety Storage 64
5.4 Accessory Limitations 66
5.5 Shape and Height 67
5.5.1 Shape 67
5.5.2 Height 69

6 Animation Variety **71**
6.1 Animation Types 71
6.1.1 Idle Animation Clips 72
6.1.2 Locomotion Animation Clips 72
6.2 Motion Kits 73
6.2.1 Data Structure 74

6.2.2 Implementation 75
6.3 Upper Body Movements 77
6.4 Complex Accessories 80

7 Real-Time Pipeline **83**
7.1 Scaler 83
7.1.1 Score Allocation 83
7.1.2 Human Instance Lists 85
7.2 Simulator 86
7.3 Animator 87
7.4 Renderer 88
7.4.1 Shadows 88
7.4.2 Virtual Human Rendering 91
7.4.3 Performance 92

8 Results and Case Studies **97**
8.1 Case Study 1: Buildings 98
8.2 Case Study 2: Theme Park 99
8.2.1 Scenario 100
8.2.2 Performance 100
8.3 Case Study 3: Ancient Pompeii 102
8.4 Case Study 4: Medical Domain 105
8.4.1 Vestibular Reeducation 105
8.4.2 CAVE 106

9 Conclusion **109**
9.1 *YaQ* Architecture 109
9.2 Appearance Variety 110
9.3 Shape Variety 111
9.4 Animation Variety. 111

10 Appendix **115**
10.1 Introduction on Navigation in *YaQ* 115
10.1.1 Navigation Graph and Semantic Model 115
10.1.2 Navigation Planning 116
10.2 Implementation of Segmentation Maps 117
10.2.1 Color Contributions 118
10.2.2 Multiple Materials 120

Remerciements

Je voudrais remercier mon directeur de thèse, Daniel Thalmann, pour m'avoir donné l'opportunité et les moyens de travailler sur cette thèse au VRlab pendant ces quatre dernières années. Un grand merci également à Boi Faltings, Stéphane Donikian, Roger Hersch, et Pierre Poulin pour accepter de faire partie de mon jury.

Sans l'aide précieuse et le support inconditionnel de Barbara Yersin, cette thèse ne serait pas la même. Merci, Barbara pour tous ces moments partagés ensemble !

Merci à Mireille Clavien, notre utilisatrice principale pour avoir testé et éprouvé nos outils, ainsi que son aide précieuse durant les deadlines.

Merci aux anciens "crowd guys" du VRlab : Julien Pettré pour nos collaborations, Sébastien Schertenleib et Pablo de Heras Ciechomski.

Merci aux membres et ex-membres du VRlab pour avoir partagé idées, discussions et bonne humeur ! Particulièrement, Ronan Boulic, Helena Grillon, Benoit Le Callennec, Damien Maupu et Daniel Raunhardt (nos quatre sujets préférés de discussions me manqueront).

Merci également aux étudiants exceptionnels avec qui j'ai eu la chance de travailler : Fiorenzo Morini, Rob van der Pol, Robin Mange et Antoine Schmid.

Merci à Josiane Bottarelli et Renaud Ott pour le support administratif et technique.

Merci au Fonds National Suisse de la Recherche Scientifique.

Pour Dina, Yaël et Alex.

Résumé

Afin de simuler des foules d'humains virtuels en temps réel et de manière réaliste, trois éléments principaux doivent être réunis. Premièrement, la *quantité*, c'est-à-dire la capacité de simuler des milliers de personnages. Deuxièmement, la *qualité*, car chaque humain virtuel dans une foule doit avoir une apparence et une animation uniques. Finalement, l'*efficacité* est primordiale, car une opération habituellement considérée comme efficace sur un seul humain virtuel devient extrêmement coûteuse quand elle est appliquée sur une grande population. Le développement d'une architecture capable de gérer ces trois aspects est un problème intéressant et stimulant que nous avons adressé dans notre recherche.

Notre première contribution est une architecture efficace et flexible, nommée YaQ, capable de simuler des milliers de personnages en temps réel. Cette plateforme, développée à l'EPFL, au VRLAB, est le résultat de plusieurs années de recherche. Elle intègre des techniques de l'état de l'art à plusieurs niveaux. YaQ a pour but de procurer des algorithmes efficaces et des solutions en temps réel pour peupler globalement et massivement des environnements à grande échelle. L'utilisation de YaQ est donc appropriée dans de nombreux domaines d'application, tels que les jeux vidéo et la réalité virtuelle. Notre architecture est spécialement efficace dans sa gestion des grandes quantités de données utilisées pour simuler des foules.

Afin de simuler de grandes foules, beaucoup de copies doivent être générées à partir d'un petit ensemble de modèles d'humains. À partir de ce point, si aucune mesure n'est prise pour varier individuellement chaque humain, de nombreux clones apparaissent dans la foule. Nous présentons plusieurs algorithmes afin de rendre chaque individu unique dans la foule. Premièrement, nous introduisons une nouvelle méthode permettant de distinguer les parties du corps d'un humain et appliquons des variétés de couleurs et motifs sur chacune d'entre elles. Deuxièmement, nous présentons deux techniques pour modifier la forme et le profil d'un humain virtuel : une méthode simple et efficace pour attacher des accessoires aux individus, ainsi que des outils pour agrandir le squelette et le maillage d'une copie. Finalement, nous contribuons également à la variété d'animations en introduisant des variations sur le haut du corps, permettant à un individu de faire un téléphone par exemple, ou d'avoir une main dans la poche, ou encore de porter des accessoires lourds.

Afin d'obtenir une grande quantité d'individus dans une foule, l'utilisation de niveaux de détail est indispensable. Nous explorons les solutions les plus adéquates pour à la fois simuler de grandes foules avec des niveaux de détail, et aussi éviter des transitions perturbantes lorsqu'un humain virtuel passe d'une représentation à une autre. Afin d'y parvenir, nous développons des solutions pour rendre la plupart des techniques de variété applicables sur tous les niveaux de détail.

Mots-clés : foules virtuelles, temps-réel, rendu, variété, animation, moteur de foules

Abstract

To simulate realistic crowds of virtual humans in real time, three main requirements need satisfaction. First of all, *quantity*, *i.e.*, the ability to simulate thousands of characters. Secondly, *quality*, because each virtual human composing a crowd needs to look unique in its appearance and animation. Finally, *efficiency* is paramount, for an operation usually efficient on a single virtual human, becomes extremely costly when applied on large crowds. Developing an architecture able to manage all three aspects is a challenging problem that we have addressed in our research.

Our first contribution is an efficient and versatile architecture called YaQ, able to simulate thousands of characters in real time. This platform, developed at EPFL-VRLab, results from several years of research and integrates state-of-the-art techniques at all levels: YaQ aims at providing efficient algorithms and real-time solutions for populating globally and massively large-scale empty environments. YaQ thus fits various application domains, such as video games and virtual reality. Our architecture is especially efficient in managing the large quantity of data that is used to simulate crowds.

In order to simulate large crowds, many instances of a small set of human templates have to be generated. From this starting point, if no care is taken to vary each character individually, many clones appear in the crowd. We present several algorithms to make each individual unique in the crowd. Firstly, we introduce a new method to distinguish body parts of a human and apply detailed color variety and patterns to each one of them. Secondly, we present two techniques to modify the shape and profile of a virtual human: a simple and efficient method for attaching accessories to individuals, and efficient tools to scale the skeleton and mesh of an instance. Finally, we also contribute to varying individuals' animation by introducing variations to the upper body movements, thus allowing characters to make a phone call, have a hand in their pocket, or carry heavy accessories, *etc.*

To achieve quantity in a crowd, levels of detail need to be used. We explore the most adequate solutions to simulate large crowds with levels of detail, while avoiding disturbing switches between two different representations of a virtual human. To do so, we develop solutions to make most variety techniques scalable to all levels of detail.

Keywords:
Virtual Crowd, Real-Time, Rendering, Variety, Animation, Crowd Engine

CHAPTER 1

Introduction

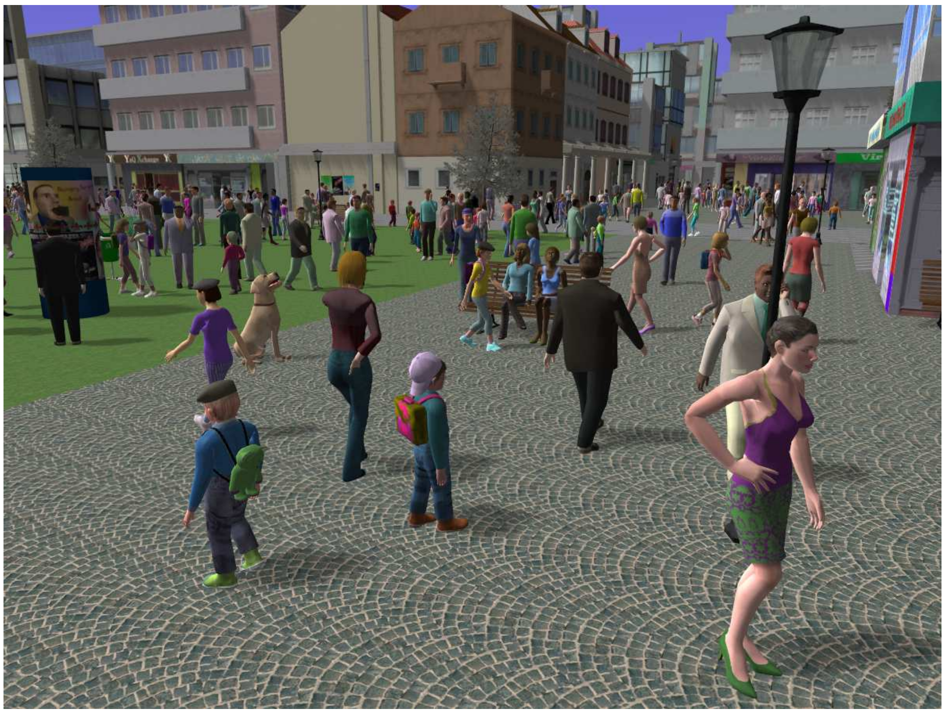


Figure 1.1: *YaQ* is a software architecture dedicated to the real-time simulation of large crowds of varied virtual humans.

Crowd simulation has a large application field ranging from architecture design to entertainment, virtual training, security, therapy for social disorders using virtual reality exposure, *etc*. Applications dedicated to urban design or security have strong needs for realism. In the Architecture domain for instance, to compare and validate designs of public places, virtual humans' navigation is simulated for the given environments. Then, at a post-processing stage the resulting locomotion trajectories are studied. In such a case, the correctness of simulation results, as compared to real humans' navigation, is obviously a crucial objective whilst performances are secondary - only reasonable computation times are required. Other applications such as video games or more generally virtual reality, require interactivity.

Interactivity means that simulation results are progressively rendered to the user, whose actions or reactions have an impact on the simulation content. As a result, the simulation is computed online, in order to account for a user's actions, and in real-time, to ensure immediate, smooth and believable rendering of the simulation content to the user. We call this type of simulation Interactive Virtual Crowds. For smooth visual output, display screen has to be refreshed at least at $25Hz$. As a result, a short computation time ($40ms$) is available between two simulation steps, whilst the number of tasks to achieve in order to refresh and render each virtual human state is high; it requires updating humans' global positions according to their goal and occurring interactions, computing their postures, updating their appearance model accordingly, and finally, rendering them to the screen. The complexity of the simulation thus directly depends on the number of humans composing the crowd. One of the most important criteria for applications requiring Interactive Virtual Crowds is the quality of the experience brought to the user, no matter the realism of the whole simulation result. Consequently, Interactive Virtual Crowds attempt to provide believable experiences to users, *i.e.*, to obtain a satisfying behavior, motion and visual appearance for virtual humans in front of the spectator, whereas those in the background or invisible areas are of lesser importance.

1.1 Motivations and Contributions

Over the last four years, we have worked on a software platform called *YaQ*, dedicated to simulation, animation and rendering of Interactive Virtual Crowds, and running on mainstream desktop computers. *YaQ* benefits from several years of research and development at EPFL-VRLab. The objective of *YaQ* is to create and animate Interactive Virtual Crowds consisting of thousands of pedestrians, in order to massively populate given environments. An example of results obtained with *YaQ* is illustrated in Figure 1.1. Our major contributions have been focused on three main aspects: first, creating the architecture of *YaQ* in order to simulate large crowds in real time. Second, exploring rendering level-of-detail techniques to display as many characters as possible. Finally, our last and main contribution consisted in exploring variety techniques in order to make each instance of a virtual human unique.

1.1.1 *YaQ* Architecture

YaQ is an engine completely dedicated to the real-time simulation of crowds. The main challenge when building such an architecture, is to be able to deal with the large quantity of information that needs to be stored, processed, and used in real time. Data have to be stored in adequate structures, so that they can be easily shared and retrieved. Also, they need to be intelligently sorted to minimize the number of context switches at runtime.

Our main contribution in this domain is the detailed mechanism of *YaQ*: it is built to tightly pack the data associated to virtual humans in order to store once information that can be shared by several instances; several lists of human ids are sorted according to different criteria at each simulation frame to accelerate the processing of the pipeline; and automatic methods to compute, serialize, and store heavy data in a dedicated database are exploited.

1.1.2 Rendering Levels of Detail

To simulate large crowds at high frame rates, it is necessary to use several levels of detail (LOD). Characters close to the camera are accurately rendered and animated with more costly methods, while those farther away are represented with less detailed, faster representations. The common process is to use many instances of a small set of human templates, *i.e.*, virtual human types identified by their mesh, skeleton, textures and LOD. There are mainly three LOD used in crowd applications, depicted in Figure 2.1: classical deformable meshes, enveloping a skeleton and skinned to perform skeletal animations, rigid meshes, which are pre-computed geometric postures of a deformable mesh, and impostors, representing a character with only two textured triangles forming a quad. Deformable meshes are altered by the online computation of their skeleton movements. Although this method is more expensive than using rigid meshes [Ulicny et al., 2004], it allows to perform special animations chosen or produced at runtime, like looking at the camera (see Figure 2.1 (left)), or mimicking facial expressions. Yet, impostors are naturally the most exploited LOD in the domain of crowds. Their main advantage is their rendering efficiency, since only two triangles per character are displayed.

1.1.3 Crowd Variety

Our main interest is focused on real-time applications where the visual uniqueness of the characters composing a crowd is paramount. On the one hand, it is required to display several thousands of virtual humans at high frame rates, using levels of detail. On the other hand, each character has to be different from all others, and its visual quality highly detailed, as illustrated in Figure 1.2.

Instantiating many characters from a limited set of human templates leads to the presence of multiple similar characters everywhere in the scene. However, the creation of an individual mesh for each character is not feasible, for it would have too high requirements in terms of design and memory. Thus, methods have to be introduced to modify each instance, so that it is visually different from all the others. Such methods also need to be scalable for all LOD used in crowd simulations to avoid inconsistencies in the individual appearances. Our main contribution is the introduction of techniques to improve the variety of crowds in three domains: visual appearance, shape, and animation.

Visual Appearance. We have developed a fast and scalable technique to obtain unique characters from a small set of basic human templates (see Figure 1.2). Using a dedicated

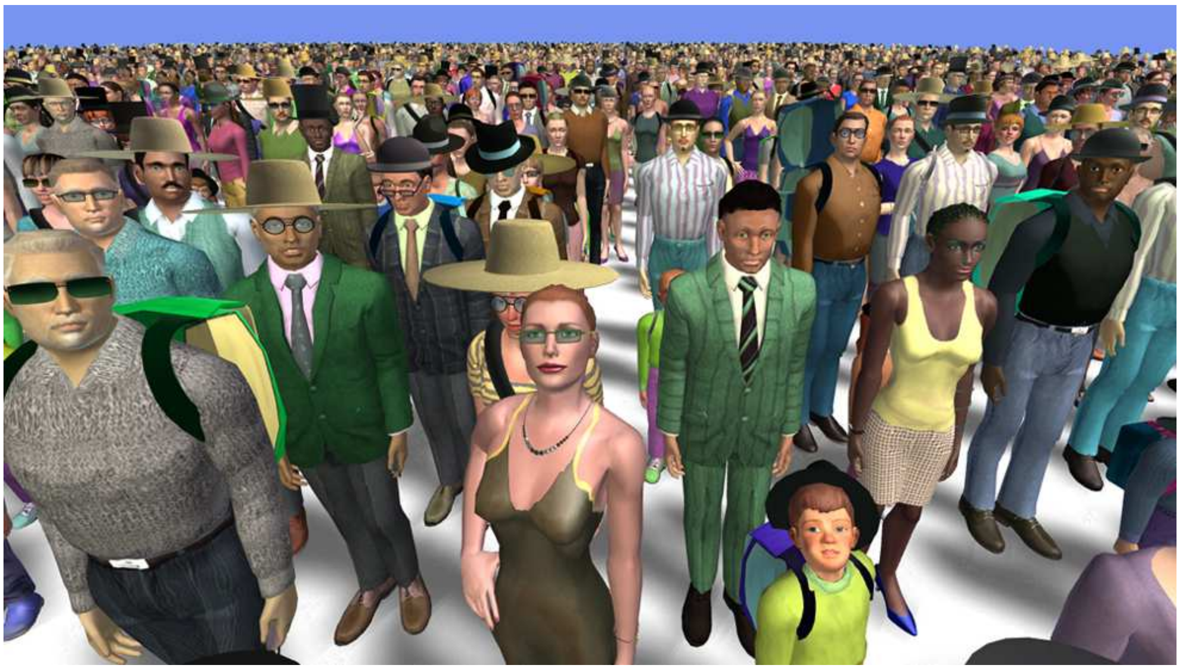


Figure 1.2: Five human templates taking full advantage of accessories and segmentation maps.

texture, called *segmentation map*, we are able to distinguish different parts of the virtual human's body, *e.g.*, skin, hair, shirt, pants. For each body part, we are able to apply a unique color, thus making two instances of a same human template visually different. As compared to previous approaches, our technique allows to have smooth transitions between body parts and to enhance character visual appearance with distinctive details, such as make-up, or fabric patterns. This method is scalable, so that all characters can be displayed consistently with any LOD used in crowd simulations.

Shape. Our contribution in the domain of shape variety is twofold. Firstly, we introduce accessories: simple meshes attached to the individuals in order to modify their profile, *e.g.*, hats, wigs, glasses, or jewelry. We distinguish two types of accessories: simple ones, that can be directly attached to a human mesh, and complex accessories, that require a modification of the human animation, *e.g.*, shopping bags, suitcases, puppets, balloons, *etc.* We have developed accessories to make them scalable: they can be used with all rendering LOD. Specifically in the case of impostors, we use a method at the pixel level to correctly place accessories, and solve occlusion issues inherent to this level of representation with a new algorithm. The additional use of accessories on top of other variety techniques allow to transform instances of a same human mesh into unique individuals. Our second contribution is to directly modify the instantiated mesh and skeleton of a human: using a dedicated design tool, it is possible to choose at which scale a skeleton can be magnified, resulting in human instances of different sizes. Also, using *fat maps*, a human instance's mesh can be deformed to result in a fat, pregnant, or thin character.

Animation. In the domain of animation, variety is also very important: if all virtual humans walk at the same speed and with the same gait, characters look closer to an army than a crowd of independent pedestrians. We have several contributions in this domain: first of all, we use a locomotion engine based on the work of [Glardon et al., 2004a,b] to generate offline many locomotion cycles at different speeds and with several gaits, using captured motion data. Second, we use a dedicated IK solver [Baerlocher and Boulic, 2004] to add to these cycles upper body variations, such as having one hand in the pocket, or on the hip, for instance. Third, further modifications of the animation are achieved online and in real time to adapt human movements to complex accessories if they wear any.

1.2 Summary of Chapters

We summarize here the topics detailed in each chapter of this document.

Chapter 2: Related Work. We start by presenting an overview of the related work achieved in the domain of real-time crowds. In this chapter, we mainly focus on previous work on crowd rendering, crowd appearance variety, and other crowd architectures.

Chapter 3: Overview. In this chapter, we present an overview of the architecture we have built: *YaQ*. We first present the main components of this structure. We also detail how the whole crowd-related data are processed and stored in *YaQ*. Finally, we shortly introduce how crowd navigation is handled. This subject is however no further detailed in this document, for it is not the topic of our thesis (see the work of [Yersin, 2009] for more details).

Chapter 4: Appearance Variety. This chapter presents our techniques to improve the appearance variety of human instances. We detail our concept of appearance sets, segmentation maps, and how they can be implemented.

Chapter 5: Shape Variety. We introduce shape variety in crowds by exploiting accessories. Their principles and implementation are detailed in this chapter. We also present techniques and tools to modify the skeleton and the mesh of each human instance.

Chapter 6: Animation Variety. In this chapter, we show the techniques used to introduce variety of animation at two levels: generating several locomotion cycles at various speeds, and modifying upper body postures both offline and online. Also, we present the motion kit, a structure built to help the animation of characters, whichever the LOD they are using.

Chapter 7: Real-time Pipeline. The pipeline of *YaQ* is here described in four steps: scaling, simulation, animation, and rendering.

Chapter 8: Results and Case Studies. This chapter is a special section that presents how some features of *YaQ* have been applied in various situations (segmentation maps applied to buildings, accessories and segmentation maps combined in a theme park simulation), and how *YaQ* as a whole has been used for several applications (cultural heritage, vestibular reeducation, agoraphobia).

Chapter 9: Conclusion. We finally present our conclusion by summarizing our contributions and introducing interesting directions for future work.

CHAPTER 2

Related Work

Crowd behavior is an intriguing subject that has been studied since the end of the nineteenth century [Bon, 1895]. The attempts to reproduce such scenes with computer simulations are however quite recent. Our crowd engine, *YaQ*, addresses three main issues in the context of real-time interactive virtual crowds: rendering large crowds using a level-of-detail approach, using color and shape variation techniques to provide each individual with a unique appearance, and making them navigate autonomously in their environment. In this chapter, we thus address the work that has been achieved in these three domains (Sections 2.1 to 2.3), and also study other crowd architectures that have been developed for full featured real-time crowd simulations in Section 2.4.

2.1 Real-time Crowd Visualization

Rendering crowds in real time is a challenging problem: all individuals need to be updated and rendered at least 25 times per second to ensure interactivity. To decrease the needed computation resources, a level-of-detail (LOD) approach is usually taken: highly detailed, but costly meshes are used to render characters close to the point of view, while lower, faster rendering methods are exploited for the larger part of the crowd, at farther distances [Ryder and Day, 2005]. Characters close to the camera are usually rendered as either one of the two following models (also illustrated in Figure 2.1):

- *Deformable meshes*, usually enveloping a skeleton and skinned to perform skeletal animations. Deformable meshes are animated online and in real time; first by computing their skeleton movements, and second, by deforming the mesh accordingly. Such computations are quite expensive, limiting the possible number of deformable meshes in a crowd. They are favored for a high level of detail, because they can perform special animations chosen or produced at runtime, like looking at the camera, or mimicking facial expressions.

- *Rigid meshes*, or pre-computed geometric postures of a deformable mesh. They have the advantage of being faster processed online than deformable meshes, since the mesh deformation is pre-computed. However, this solution has higher memory requirements and does not allow for procedural animations.

In some cases, levels of detail were directly applied to the number of polygons composing the human meshes. For instance, De Heras Ciechomski *et al.* [Ciechomski et al., 2004] used rigid meshes only, and reduced the number of triangles of the meshes with the increasing distance to the camera. As for the rendering of the massive, low-resolution parts of the crowd, we distinguish two approaches: the point-based approach, and the image-based approach.

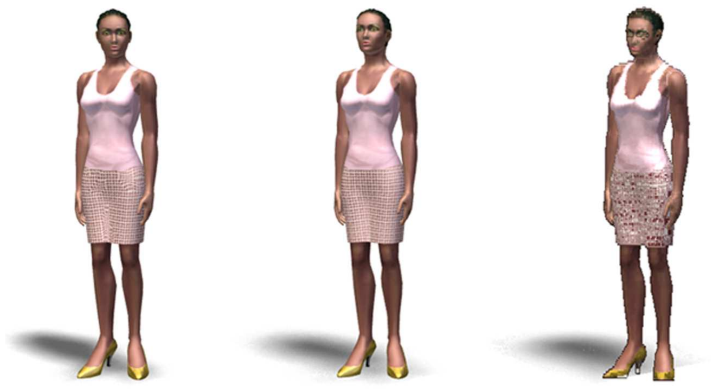


Figure 2.1: *(left to right)* A deformable mesh, skinned and animated in real time; a rigid mesh, *i.e.*, pre-computed posture of the same mesh; and an impostor, a pre-computed image of the same mesh, textured onto a quad to give the illusion of a 3D mesh.

2.1.1 Point-based Approach

In 1985, Levoy and Whitted first had the idea of using a point-based technique: they decoupled the modeled geometry from the rendering process by using points as a meta-primitive. This meta-primitive plays the role of a mediator between the traditionally modeled geometry and the rendering pipeline: geometrical objects are first converted into points (the meta-primitive), to be rendered on the screen [Levoy and Whitted, 1985]. Wand and Strasser presented a multi-resolution rendering algorithm based on this approach to render large crowds of animated characters: having for sole input the keyframe animations of meshes, a hierarchy of point samples and triangles is built to represent the various resolutions of the scene [Wand and Straßer, 2002]. Their results are illustrated in Figure 2.2. Rudomin and Millan later combined point-based rendering for distant characters with displaced subdivision surfaces for characters at a closer range [Rudomin and Millan, 2004].

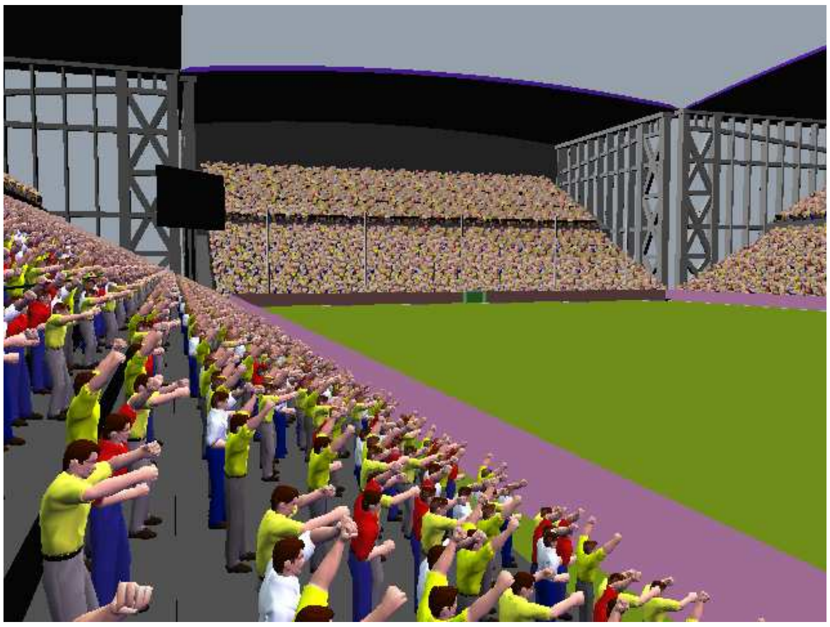


Figure 2.2: Results obtained in [Wand and Straßer, 2002], using a hierarchy of points and triangles.

2.1.2 Image-based Approach

Combining an image-based technique with highly detailed meshes to render large crowds in real time is the most common approach. The quality of a character appearance is usually very high when close to the camera, using a deformable or rigid mesh, and degrades over the distance, until it becomes an animated *impostor*. Impostors are sets of 2D pre-computed images of discrete keyframes of animated virtual humans: images are pre-computed using various view angles in order to fit any relative position between the camera and humans. These images are then exploited at runtime in place of 3D models when rendering the scene. Impostors can thus represent a character in the scene with only two triangles forming a quad, and textured with one of these images. An example of such an image is presented on the right-side of Figure 2.1. The main advantage of impostors is their rendering efficiency, since only two triangles per character are displayed. Their major drawback is their memory requirements, higher than the ones of rigid meshes.

Aubel *et al.* first applied the concept of impostors to virtual humans [Aubel et al., 2000]. Tecchia *et al.* used the same concept to render large crowds of varied characters [Tecchia and Chrysanthou, 2000]. They reduced the impostor memory consumption first by working with symmetrical animations, thus limiting the number of images required for each keyframe (images can be mirrored), and second, by tightly packing the images together, removing all empty space around each of them [Tecchia et al., 2002b,a].

Later, Dobbyn *et al.* presented the first hybrid approach, combining impostors with rigid meshes in [Dobbyn et al., 2005]. Rigid meshes were used at the forefront, replaced by impostors whenever the distance to the point of view became too large. To ensure no disturbing popping artifacts when switching from a level of detail to another, a "pixel to texel" ratio was enforced. The results they obtained is illustrated in Figure 2.3. Following this work, several

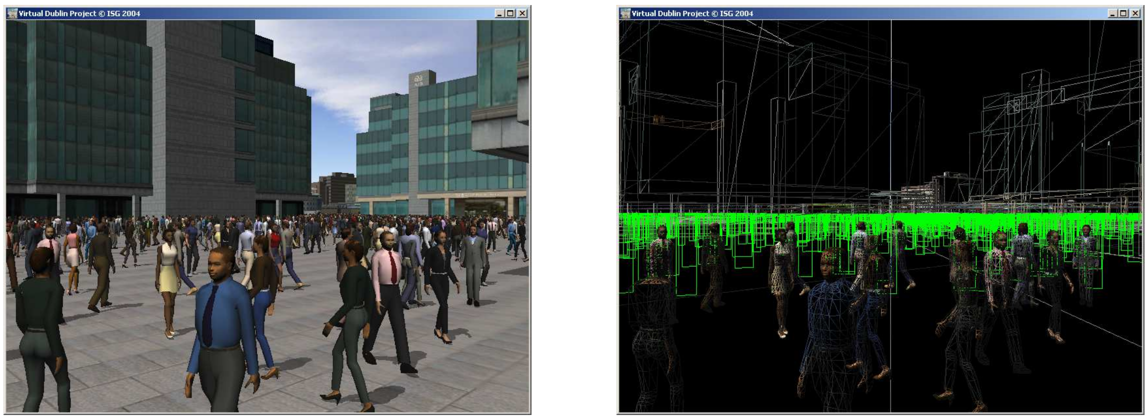

Virtual Dublin Project © ISG 2004
Virtual Dublin Project © ISG 2004

Figure 2.3: Dobbyn et al. combined rigid meshes with impostors to render large crowds and keep a high appearance quality close to the camera [Dobbyn et al., 2005].

studies have been conducted in order to determine which LOD is the most adequate, and at which distances LOD switching should be achieved to avoid popping artifacts [Hamill et al., 2005; McDonnell et al., 2005]. More recently, Millan and Rudomin combined impostors and instanced geometries to render very large crowds. Their instancing approach has the advantage of maximizing the use of the GPU, however it limits the number of possible animations too [Millan and Rudomin, 2006]. *YaQ* also benefits from a level-of-detail strategy. We use three levels: at the forefront, highly detailed deformable meshes capable of facial and hand animation are used. Then, at a farther distance, pre-computed rigid meshes are displayed, and finally, when characters appear very small, impostors are used [Maïm et al., 2009]. In Figure 2.1, we illustrate these three LOD. We further detail our approach in Chapter 3.

Impostors being memory hungry, a new and efficient level of detail has been introduced by Kavan *et al*. A polypostor represents a virtual human with a small set of 2D polygons, animated by displacing their vertices [Kavan et al., 2008]. This representation offers much lower memory consumption, since only one texture per virtual human needs to be stored, and each animation corresponds to a series of vertex displacements. An image of an animated polypostor is showed in Figure 2.4.

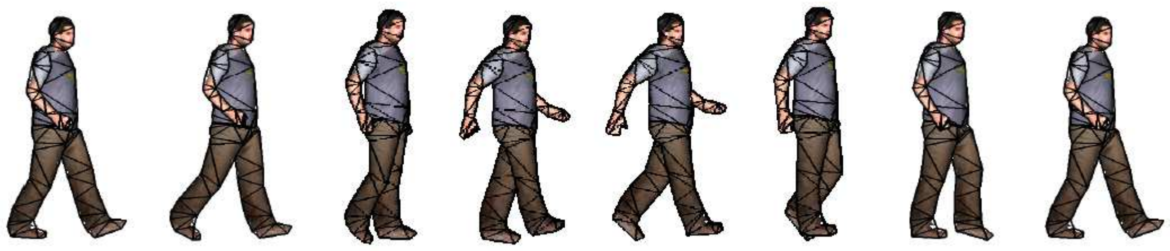


Figure 2.4: A polypostor is a 2D polygonal representation of a virtual human. Its animation is only based on the displacement of its vertices [Kavan et al., 2008].

Recently, Barczak *et al.* presented a solution entirely programmed on the GPU to render large crowds with three GPU-based levels of detail: close to the point of view, characters are rendered with hardware tessellation and displacement mapping; at a reasonable distance, a conventional rendering approach is used; at a far distance, simplified shaders and geometry are exploited [Barczak et al., 2008].

Another issue inherent to impostors is to solve occlusions when two of them intersect. Indeed, impostors being rendered as a quad, or two triangles, accurate occlusion between them if they are at the same depth results in disturbing artifacts. Schaufler introduced nailboards, able to overcome visibility artifacts by storing small depth offsets in the alpha channel of their texture [Schaufler, 1997]. However, this method is limited to orthographic or near-orthographic views. Aubel *et al.* also introduced a technique to avoid visibility issues by dividing an impostor into a series of body parts, *i.e.*, several quads, each assigned with a specific depth value [Aubel et al., 2000]. This technique is suitable when used on body parts only, but it is not adapted for subtle cases, where pixels need to be processed individually. Kavan *et al.* order the body parts of a polypostor by creating a visibility graph, based on the 3D model. In Chapter 5.2, we present a GPU-based per-pixel approach, allowing to solve occlusion problems at the pixel level, such as positioning the impostor of a backpack and its straps on the shoulders of a virtual human impostor.

2.2 Crowd Appearance Variety

Rendering large crowds is usually based on the instantiation of a small set of *human templates*. A human template is a virtual human type, defined by three main components:

- At least one mesh, usually composed of triangles. From a first high-resolution mesh, it is possible to derive several meshes at lower resolutions (less triangles) and use them as levels of detail.
- A unique skeleton, that defines the human template's joints and their size. The mesh is skinned around the skeleton to enable animation.
- A texture, mapped onto the mesh with UV coordinates.

If such templates are instantiated many times, large crowds can be created. However, all instances of a same template have exactly the same look. For this reason, several approaches have tried to add variety to instances of a same template. In the following sections, we distinguish two approaches: one focused on color variety of instances, and another concentrated on shape variety.

2.2.1 Color Variety

In a recent study, McDonnell *et al.* showed that introducing color variety was paramount to avoid a clone effect [McDonnell et al., 2008]. A first step to improve variety is to create several textures per template, and use them randomly at instantiation. Although this solution offers the best results, designers cannot create one texture for each instance for obvious reasons. Previous work on color variety is based on the idea of dividing a human template into several body parts, identified by specific intensities in the alpha channel of the template texture, as illustrated in Figure 2.5. At runtime, each body part of each character is assigned a color in order to modulate the texture. Tecchia *et al.* used several passes to render each impostor body part [Tecchia et al., 2002b,a]. Dobbyn *et al.* extended the method and avoided multi-pass rendering using programmable graphics hardware [Dobbyn et al., 2005].

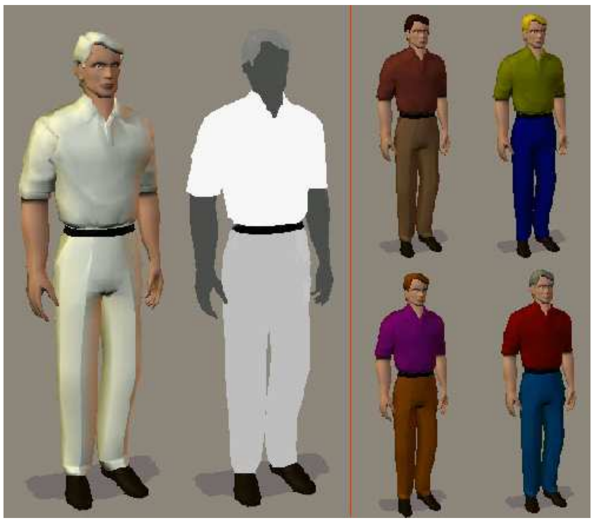


Figure 2.5: Variety in appearance is achieved by dividing a human into several body parts and coloring each part with varied colors. The body parts are differentiated by a specific intensity in the alpha channel of the texture [Tecchia et al., 2002a].

Based on the same idea, Gosselin *et al.* showed how to vary characters sharing the same texture by changing their tints. They also presented a method to selectively add decals to the characters' uniforms [Gosselin et al., 2004]. However, their approach is only applied to armies of similar characters, and the introduced differences are not sufficient when working with crowds of civilians. An illustration of their results is showed in Figure 2.6. Galvao *et al.* presented another solution to obtain color variety without using the alpha channel, which can then be exploited for other purposes [Galvao et al., 2008]. In Chapter 4, we present our approach to obtain improved color variety, using segmentation maps. With segmentation maps, it is possible to obtain much more subtle effects, like make-up, freckles, cloth patterns, *etc.* [Maïm et al., 2009].

2.2.2 Shape Variety

A second approach to further vary individuals in a crowd is to modify the shape of instances. When we use the term "shape modification", we englobe the work achieved to directly modify the mesh of an instance, but also the research that has been conducted in the domain of cloth simulation for crowds, and the addition of accessory meshes.

One of the basic approaches to generate new human meshes is to morph between existing models [Lee and Magnenat-Thalmann, 2001]. To make such an operation intuitive, one solution is to extract high-level parameters from the existing models. Then, new meshes are created by providing a user-defined set of parameters, which will define how to morph the models [Seo et al., 2003; Allen et al., 2004]. Another solution is to use anthropometric data

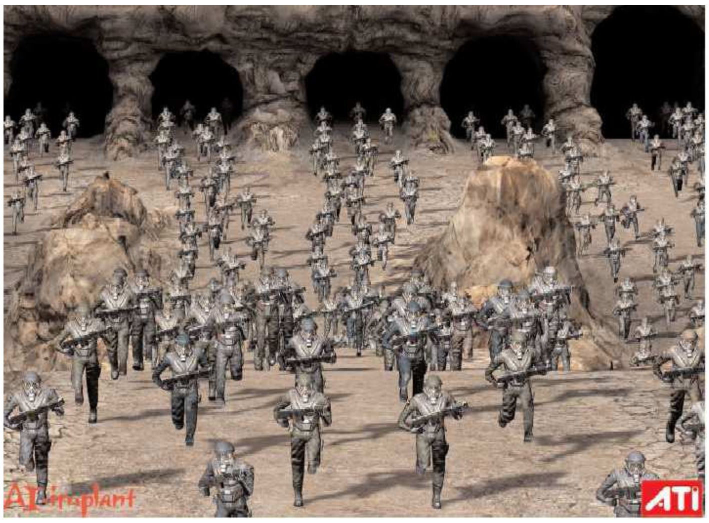


Figure 2.6: Crowd using color variety and decals from [Gosselin et al., 2004].

to correctly choose how a human mesh can be resized. Seo *et al.* proposed such an approach to generate virtual populations: the generation of human bodies is based on the kinematic properties of anthropometry (size, shape, proportions) [Seo et al., 2002]. Unfortunately, the generation process was slow, and crowds were thus limited to a few dozen characters. Kasap and Magnenat-Thalmann [Kasap and Magnenat-Thalmann, 2007] presented an improved technique, also based on anthropometric parameters. They divide the body into segments, and deform each of them with freeform deformation methods and radial functions, while preserving skinning information. More recently, Galvao *et al.* presented a solution to vary the shape of humans composing a crowd: they apply variety at different levels: body mass, limb size, and muscle bulge. Body mass and muscle bulge variety is achieved by combining weighted displacement maps. The limb size and character's height are modified using a skeleton displacement tool: scaling values are associated to each joint. The mesh follows the skeleton deformations, according to its skinning properties [Galvao et al., 2008]. The results obtained with their work is illustrated in Figure 2.7.

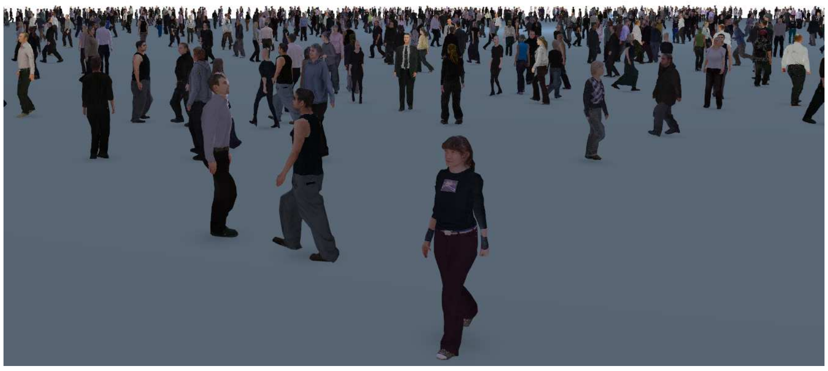


Figure 2.7: Results obtained in [Galvao et al., 2008] by varying the color, limb size, muscle bulge, and body mass of characters.

To modify the shape of a virtual human, it is also possible to attach various small meshes to its body. Recently, Dudash demonstrated in an NVidia white paper how to divide a human mesh into several pieces and use these pieces to recompose varied meshes [Dudash, 2007]. In his example, several warriors were divided according to their armor pieces and weapons.

Finally, clothes simulation is another approach to further vary characters [Ryder and Day, 2005], but for real-time crowd applications, their animation remains too expensive to be used. Dobbyn *et al.* proposed to pre-simulate cloth mesh deformation and use the resulting animation at runtime on impostors (see Figure 2.8) [Dobbyn et al., 2006]. The same year, McDonnell *et al.* presented a perceptual study of virtual humans wearing deformable clothing at different levels of detail [McDonnell et al., 2006].

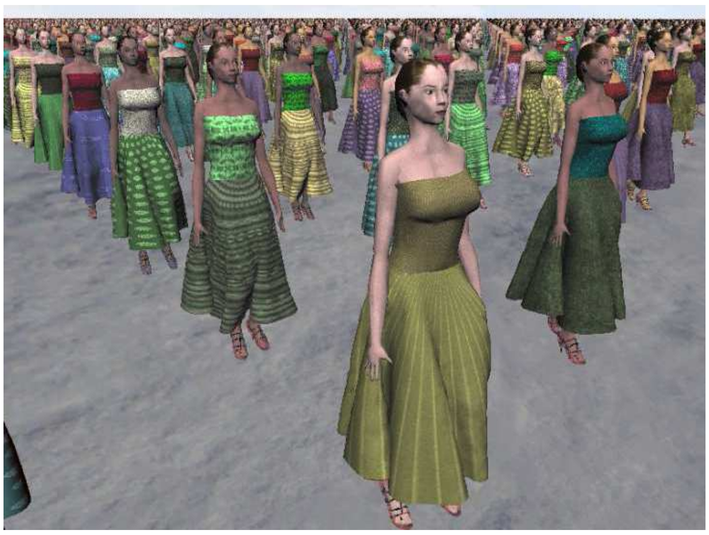


Figure 2.8: The physical simulation of clothes is pre-computed and re-used on rigid meshes and impostors [Dobbyn et al., 2006].

2.3 Crowd Motion Planning and Navigation

The first studied approach, *i.e.*, agent-based, represents a natural way to simulate crowds as independent individuals interacting with each other. Such algorithms usually handle short distance avoidance, and navigation remains local. Reynolds proposed to use simple rules to model crowds of interacting agents [Reynolds, 1987, 1999]. Heigeas *et al.* introduced a physically-based interactive particle system to model emergent crowd behavior often encountered in emergency situations [Heigeas et al., 2003]. Kirchner and Shadschneider used static potential fields to rule a cellular automaton [Kirchner and Shadschneider, 2001]. Nevertheless, the main problem with agent-based algorithms is their low performance. With these methods, simulating very large crowds in real time is unfeasible without distributing the workload on parallel processors [Reynolds, 2006]. Moreover, such approaches forbid the construction of autonomous adaptable behaviors, and can only manage crowds of pedestrians with local objectives.

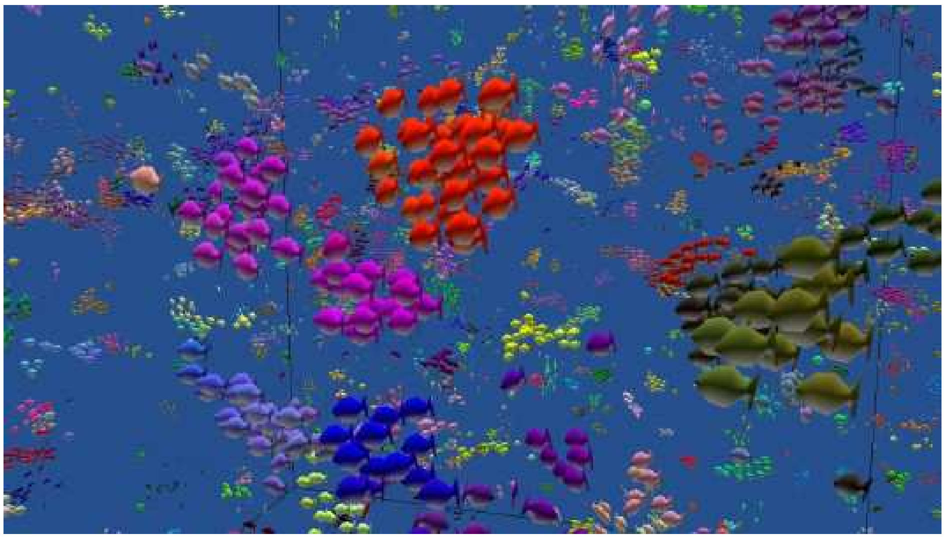


Figure 2.9: Reynolds takes advantage of the parallel processors of the PS3 to simulate large crowds of fishes with an agent-based method [Reynolds, 2006].

To solve the problems inherent in local navigation, some behavioral approaches have been extended with global navigation. Sung *et al.*, given constraints at specific time intervals on character poses, positions, orientations, used a motion graph to generate the adequate motion [Sung et al., 2005]. Lau and Kuffner used pre-computed search trees of motion clips to accelerate the search for the best paths and motion sequences to reach an objective [Lau and Kuffner, 2006]. Lamarche and Donikian used automatic topological model extraction of the environment for navigation [Lamarche and Donikian, 2004]. Their results for outdoor navigation is illustrated in Figure 2.10. Although these approaches offer appealing results, they are not fast enough to simulate thousands of pedestrians in real time. Pettré *et al.* presented the *navigation graph*, a structure automatically extracted from an environment geometry, allowing to solve global path planning requests [Pettré et al., 2006; Pettré et al., 2007]. The main advantage of this technique is that it handles uneven and multi-layered terrains. Nevertheless, it does not treat inter-pedestrian collision avoidance. Finally, Helbing *et al.* used agent-based approaches to handle motion planning, but mainly focused on emergent crowd behaviors in particular scenarii [Helbing et al., 1994, 2000] .

Another approach for motion planning is inspired from fluid dynamics. Such techniques use a grid to discretize the environment into cells. Hughes used density fields to steer pedestrians toward their goals and avoid collisions [Hughes, 2002, 2003]. Chenney used flow tiles to represent small stationary regions of velocity fields that can be pieced together to drive crowds [Chenney, 2004]. More recently, Treuille *et al.* used a dynamic potential field to represent the best path to a goal. Pedestrians are steered according to the potential gradient, avoiding collision with the environment and other pedestrians (see Figure 2.11) [Treuille et al., 2006]. Fluid dynamics represent an interesting solution in applications where the lack of individuality of each pedestrian is unimportant. Indeed, these solutions are usually meant to steer large groups of avatars towards a shared goal.

Recently, a new branch of research tries to extract patterns from real pedestrian avoidance behaviors in order to rule their models, [Lerner et al., 2007; Paris et al., 2007; Lee et al.,

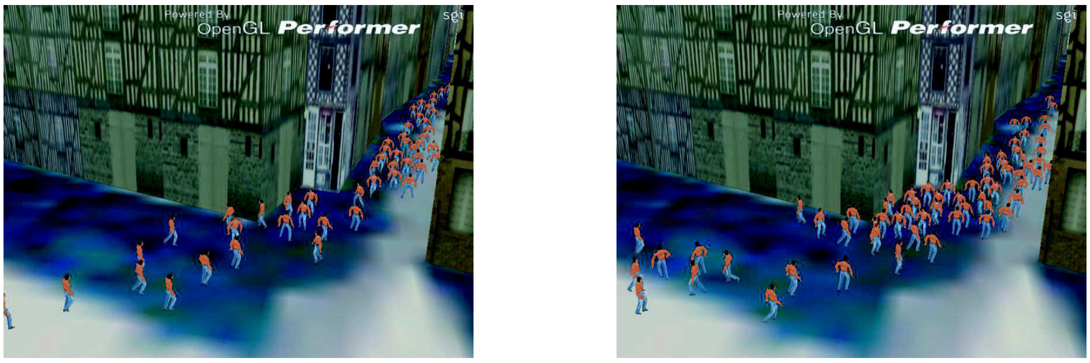


Figure 2.10: Outdoor navigation of pedestrians in [Lamarche and Donikian, 2004]. Path planning is based on the topological properties of the environment, while inter-pedestrian avoidance is ruled by a neighborhood graph.

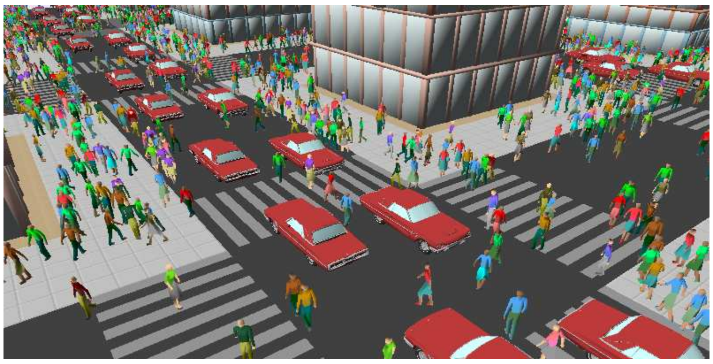


Figure 2.11: In [Treuille et al., 2006], crowds are steered by a potential field to reach their goals while avoiding collisions.

2007]. As of today, the obtained performance does not fit to large interactive crowds.

YaQ uses an architecture with levels of detail to plan the motion of large crowds. Based on a navigation graph, we divide the environment into regions of varying interests. In regions of high interest, we exploit a potential field-based approach. Since we only use it locally, we can plan motion for many more groups and with finer grid cells than with an algorithm purely based on it. In other regions, motion planning is ruled by the navigation graph and short-term collision avoidance algorithms. Our local use of a potential field-based approach allows us to plan motion for many more groups and with finer grid cells than with a purely potential field algorithm.

2.4 Crowd Architectures

Similarly to *YaQ*, there exist some architectures able to handle all aspects of crowds, *i.e.*, rendering, motion planning, behavior, and animation.

Musse *et al.* presented *ViCrowd*, a hierarchical model to simulate crowds in real time in collaborative virtual environments. Their platform combines LOD techniques for the geometry display and animation [Musse et al., 1998] and group-based intelligent behaviors [Musse and Thalmann, 2001]. The Agent Behaviour Simulator (*ABS*) is another behavioral model dedicated to real-time crowd simulation. The pedestrians are rendered with an image-based technique (see Section 2.1) [Tecchia and Chrysanthou, 2000], while their navigation and behavior are ruled by 2D maps [Tecchia et al., 2001]: a map to detect inter-pedestrian collisions, a map to detect collisions with the environment, and two maps to rule the pedestrians' simple and complex behaviors (see Figure 2.12).

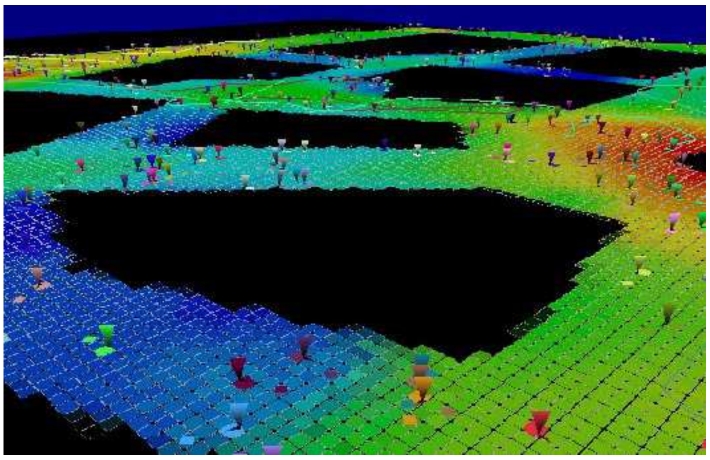


Figure 2.12: ABS: the Agent Behavior Simulator, rules pedestrian behaviors based on several 2D maps of the environment [Tecchia et al., 2001].

Giang *et al.* and O'Sullivan *et al.* developed *ALOHA* (Adaptive Level Of Detail for Human Animation), a framework applying a LOD approach to the geometry, motion, and behavior of crowds [Giang et al., 2000; O'Sullivan et al., 2003]. Unlike the usual approach, the geometry LODs of ALOHA are created in a bottom-up approach: the process starts with a low-resolution mesh and progressively improves the appearance of the virtual human using subdivision surfaces. As for the motion, characters play keyframed animations at a low level, and use a reaching-and-grasping system for higher LODs. Behavior LODs are handled differently, depending on the situation: conversational LODs are used when characters are socializing and talking/answering to each other (see Figure 2.13), while Artificial Intelligence LODs are developed for other cases.

Sung *et al.* presented a scalable crowd simulator that provides visually convincing crowds in terms of rendering, animation, and behavior. Their system has two levels: at a high level, a controller takes care of local behavior of agents, based on their position in space and what is happening in their close neighborhood. At a lower level, a probabilistic finite state machine decides of the pedestrians actions [Sung et al., 2004], the crowd animation is based on a Snap Together Motion process [Gleicher et al., 2008]. Shao and Terzopoulos presented an agent-based model to simulate crowds [Shao and Terzopoulos, 2005]. The rendering engine is adapted from a commercial software package, and their main contribution is on the behavioral level. Their approach is bottom-up: reactive behaviors are used as building blocks to support more complex motivational behaviors. The decision-making of pedestrians is controlled by an action selection mechanism. More recently, Pelechano *et al.* presented

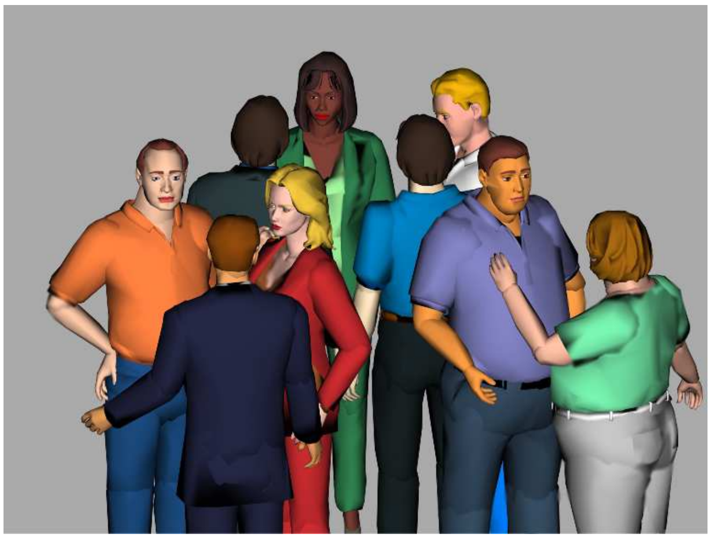


Figure 2.13: Group of characters socializing in [O'Sullivan et al., 2003].

the *HiDAC* (High-Density Autonomous Crowds) system. Although they do not detail how the rendering and animation of pedestrians is performed, they display realistic characters in order to be able to simulate densely crowded places, where virtual humans may push each other, fall, *etc.* Behaviors are determined for each agent individually, with a two-level approach: at a high level, navigation, learning, communication and decision-making are handled [Pelechano and Badler, 2006], and at a lower level, perception of the environment and reactive behaviors allow to handle collision avoidance [Pelechano et al., 2007].

CHAPTER 3

Overview

In this chapter, we present an overview of *YaQ*, our crowd engine. We first provide a general insight on its architecture in Section 3.1. In Section 3.2, we fully detail the main input of *YaQ*: the human template and its different rendering levels of detail. Finally, in Section 3.3, we present how the large amount of data necessary to simulate a crowd is handled and stored by *YaQ*.

3.1 YaQ Architecture

As illustrated in Figure 3.1, *YaQ* architecture is composed of two offline and one online elements: *Variety*, *Navigation*, and *Real-time*.

Variety. When displaying crowds, on the one hand, each character needs to get a unique appearance (see Figure 3.2) in order to simulate individuality. On the other hand, it is both fastidious to design every single character and to store its appearance model: this would result in unconceivable memory consumption and design time. Our solution is to start with a limited set of human templates, from which we apply three different types of variations to create thousands of unique instances. The process is schematized in green in Figure 3.1. We later devote one chapter to each of these types:

- In Chapter 4, we detail the techniques used to modify the colors and shading of virtual humans, *e.g.*, their clothes, skin, hair.
- Chapter 5 focuses on methods that help changing the shape of a virtual human; for instance, accessorizing characters, or making them taller/smaller, *etc*.

- The last type of variety we introduce for crowds is the individualization of their animations. To simulate a crowd in *YaQ*, many locomotion animations at various speeds are generated in a pre-process with a dedicated engine [Glardon et al., 2004a,b]. Other, more quiet animations, such as standing, talking, or sitting, are hand-designed. In order to animate humans whatever their rendering level of detail (LOD), we have created a dedicated structure, called *motion kit*. A motion kit contains the information to play an animation at any LOD. Thus, instead of associating an animation to each human instance, we associate a motion kit, which provides the adequate data, whichever the LOD of the instance. Our strategy to handle animations in *YaQ*, to vary them, and to create motion kits is explained in Chapter 6.

Navigation. This component of *YaQ* takes care of structuring the environment to plan the motion of the whole crowd. The structure we use is based on a dedicated cell-decomposition technique called navigation graph [Pettré et al., 2006; Pettré et al., 2007]: from a 3D model of a scene, a navigation graph is automatically derived. It captures and models both the geometry and the topology of the navigable space. The navigation graph is used as a basic structure to categorize the environment into regions of various interest. With the same concept of levels of detail we use for rendering, regions of high/medium/low interest are identified in the environment, and ruled by different motion planning algorithms. Each algorithm, in its own way, provides waypoints to steer virtual humans in real-time towards their goals. A second interesting aspect of navigation graphs is to use them for triggering situation-based behaviors. Using a semantic model of the environment (see Figure 3.1), the navigation graph is augmented with semantic data to develop crowd behaviors: specific actions are triggered in desired areas of the environment. We provide an overview on the Navigation component in Figure 3.1 (in red). Also, a short introduction on this subject is available in Section 10.1 of the *Appendix*. For a more detailed presentation of this component, the reader is advised to refer to the work of [Yersin, 2009].

Real-time. Once the Variety and Navigation components have computed the instances and the waypoints to navigate them, the Real-Time component can start the simulation. Each frame of simulation is separated into four different stages, illustrated in blue in Figure 3.1. The first stage, the Scaler, assesses the importance of the different areas of the screen, *i.e.*, it decides of the level-of-detail strategy for the current frame. Second, the Simulator steers the instances toward their next waypoint, while avoiding collisions between them. The Animator modifies the posture of the instances to reflect the locomotion or other action changes. Finally, the Renderer efficiently displays the varied instances and the environment, as well as their projected shadows, to the screen. The Real-time component is detailed in Chapter 7.

3.2 Virtual Human Representations

In an ideal world, graphic cards would be able, at each frame, to render an infinite number of triangles with an arbitrary complex shading on them. To visualize crowds of virtual humans, we would simply use thousands of very detailed meshes, *e.g.*, capable of hand and facial

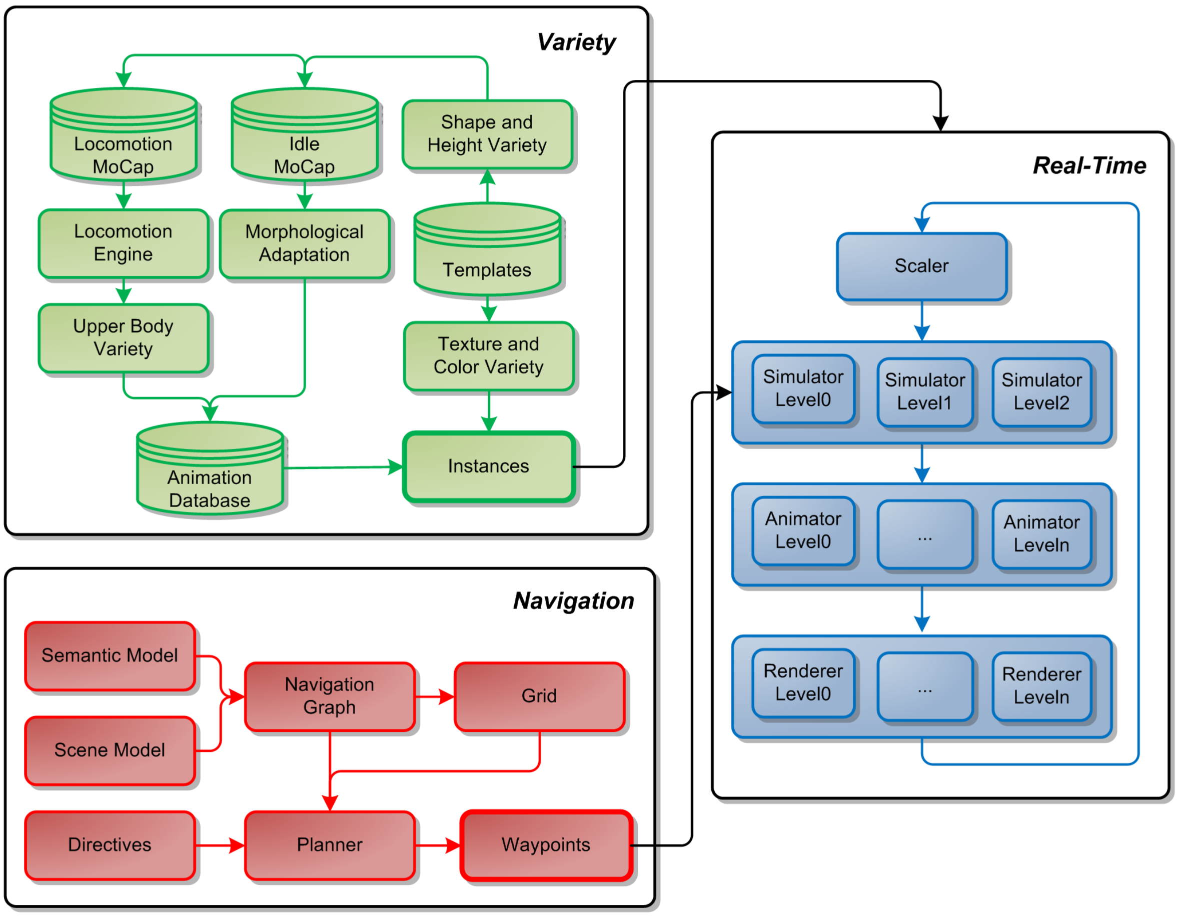


Figure 3.1: The architecture of YaQ is divided into three components. In a first step, the Variety component (in green) and the Navigation component (in red) pre-compute the necessary data (instances and waypoints). In a second step, these data are fed to the Real-time component (in blue).

animations. Unfortunately, in spite of the recent programmable graphics hardware advances, we are still compelled to stick to a limited triangle budget per frame. This budget is spent wisely to be able to display dense crowds without too many perceptible degradations. The concept of levels of detail (LOD), extensively treated in the literature [Luebke et al., 2002], is exploited to meet our real-time constraints. For a crowd of virtual humans specifically, and depending on the location of the camera, a character is rendered with a particular representation, resulting from the compromise of rendering cost and quality. In this section, we first introduce the data structure we use to create and simulate virtual humans: the human template. Then, we describe the three levels of detail a human template uses: the deformable mesh, the rigid mesh, and finally the impostor.

3.2.1 Human Template

A type of human such as a woman, man, or child is described as a human template, which consists of :

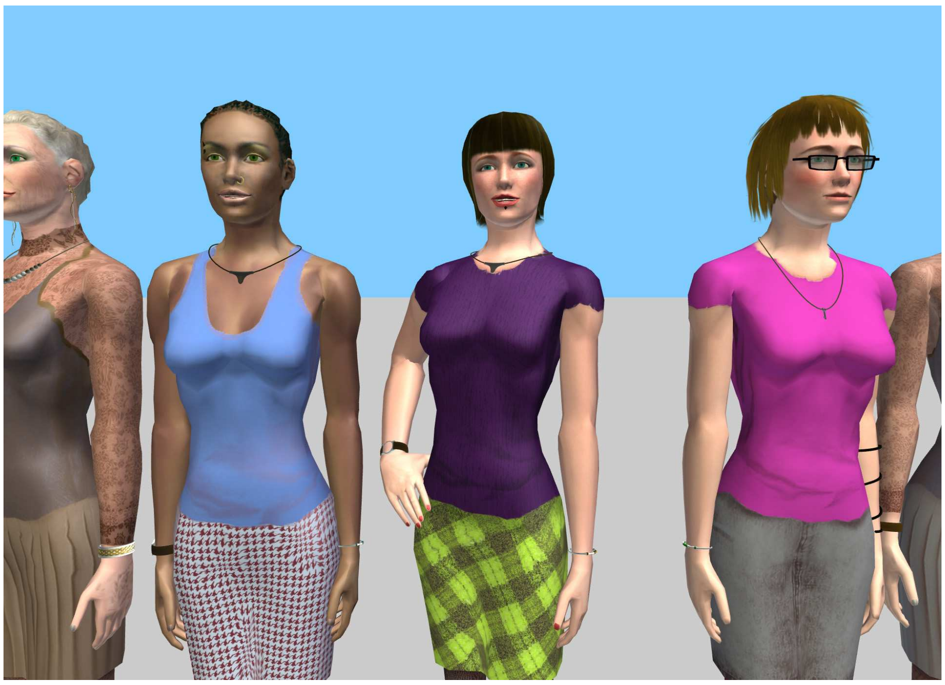


Figure 3.2: By applying variety techniques at three levels, instances of a same template seem unique. Color variations are used to create clothes and skin patterns, shape variations are introduced with accessories, such as wigs or glasses, and various animations are exploited for further individualization.

- A skeleton, composed of 86 joints, representing articulations;
- A set of meshes, all respresenting the same virtual human, but with a decreasing number of triangles;
- Several appearance sets, used to vary its appearance;
- A set of animation sequences which it can play.

Each rendered virtual human is derived from a human template, *i.e.*, it is an instance of a human template. In order for all the instances of a same human template to look different, we use several appearance sets, that allow to vary the texture applied to the instances, and modulate the colors of the texture (see Chapter 4).

3.2.2 Deformable Mesh

A deformable mesh is a representation of a human template composed of triangles. In *YaQ*, it is enveloping a skeleton, used for animation: when the skeleton moves, the vertices of the mesh follow smoothly its joint movements, similarly to our skin [Magnenat-Thalmann et al., 1988]. We call such an animation a *skeletal animation*. Each vertex of the mesh is influenced by one or a few joints. Thus, at every keyframe of an animation sequence, a vertex is deformed by the weighted transformation of the joints influencing it. The corresponding equation is:

$$v(t) = \sum_{i=1}^{n} \chi_i^t \chi_i^{-ref} v^{ref} \tag{3.1}$$

where $v(t)$ is the deformed vertex at time t, influenced by n joints, χ_i^t is the global transform of joint i at time t, χ_i^{-ref} is the inverse global transform of the joint in the reference position, and v^{ref} is the vertex in its reference position. This technique is known as skeletal subspace deformation, or skinning.

The skinning can be efficiently performed by the GPU: the deformable mesh sends the joint transformations of its skeleton to the GPU, that takes care of moving each vertex according to its joint influences. However, it is important to take into account the limitations of graphic cards (Shader Model 2 & 3 [nvidia, 2006]), that can store only up to 256 atomic values, *i.e.*, 256 vectors of four floating points. The joint transformations of a skeleton can be sent to the GPU as 4×4 matrices, *i.e.*, four atomic values. This way, the maximum number of joints a skeleton can have reaches:

$$\frac{256}{4} = 64. \tag{3.2}$$

When wishing to perform hand and facial animations, 64 joints are not sufficient. Our solution is to send each joint transformation to the GPU as a unit quaternion and a translation, *i.e.*, two atomic values. This allows to double the number of joints possible to send. Note that one usually does not wish to use all the atomic structures of a GPU exclusively for the joints of a skeleton, since it is usually exploited to process other data.

Rendering deformable meshes is very costly, due primarily to a pipeline flush occuring each time a new virtual human is rendered, and also to the expensive vertex skinning and joint transmission. Nevertheless, it would be a great quality drop to do without them, indeed:

- They are the most flexible representation to animate, allowing even for facial and hand animation (if using a sufficiently detailed skeleton).
- Such animation sequences, called skeletal animations, are cheap to store in memory: for each keyframe, only the transformation of deforming joints, *i.e.*, those moved in the animation, need to be kept. Thus, a tremendous quantity of those animations can be exploited in the simulation, increasing crowd movement variety.
- Procedural and composited animations are suited for this representation, *e.g.*, look at the camera, or on-the-fly idle motion generation (see for example Egges *et al.* [Egges et al., 2006]).
- Blending is also possible for smooth transitions between different skeletal animations.

Unfortunately, the use of deformable meshes as the sole representation of virtual humans in a crowd is too prohibitive. We therefore use them in a limited number and only at the forefront of the camera. Note that before switching to rigid meshes, we use several deformable meshes, keeping the same animation algorithm, but with a mesh of a decreasing number of triangles.

Skinned and textured deformable meshes require skilled designers. But once finished, they are automatically used as the raw material to derive all subsequent representations: the rigid meshes and the impostors.

3.2.3 Rigid Meshes

A rigid mesh is a precomputed geometric posture of a deformable mesh, thus sharing the very same appearance. A rigid animation sequence is always inspired from an original skeletal animation, and from an external point of view, both look alike. However, the process to create them is different. To compute a keyframe of a rigid animation, the corresponding keyframe for the skeletal animation is retrieved. It provides a skeleton posture (or joint transformations). Then, in a preprocess, each vertex is deformed on the CPU, as opposed to a skeletal animation, where the vertex deformation is achieved online, and on the GPU. Once the rigid mesh is deformed, it is stored as a keyframe, in a table of vertices, normals (3D points), and texture coordinates (2D points). This process is repeated for each keyframe of a rigid animation. At runtime, a rigid animation is simply played as a succession of several postures or keyframes. There are several advantages in using such a representation:

- It is much faster to display, because the skeleton deformation and vertex skinning stages are already done and stored in keyframes. The communication between the CPU and the GPU is kept to a minimum, since no joint transformations need to be sent, and pipeline flushing is significantly reduced.
- It looks exactly the same as the skeletal animation used to generate it.

The gain in speed brought by this new representation is considerable. It is possible to display about 10 times more rigid meshes than deformable meshes (see Section 7.4.3 for detailed results). However, the rigid meshes need to be displayed farther from the camera than deformable meshes, because they allow for neither procedural animations, nor blending, and no composited, facial, or hand animation are possible either.

3.2.4 Impostor

An impostor is the less detailed representation, and extensively exploited in the domain of crowd rendering [Tecchia et al., 2002a; Dobbyn et al., 2005; Millan and Rudomin, 2006]. An impostor represents a virtual human with only two textured triangles, forming a quad, which is enough to keep the desired illusion at long range from the camera. Similarly to a rigid animation, an impostor animation is a succession of postures, or keyframes, inspired from an original skeletal animation. The main difference with a rigid animation is that it is only a 2D image of the posture that is kept for each keyframe, instead of the whole geometry. Creating an impostor animation is complex and time consuming. Thus, its construction is achieved in a preprocess, and the result is then stored into a database in a binary format (see Section 3.3.2), similarly to a rigid animation. We detail here how each keyframe of an impostor animation is developed. The first step when generating such a keyframe for a human template is to create two textures, or atlas:

- A normal map, storing in its texels the 3D normals as RGB components. This normal map is necessary to apply the correct shading to the virtual humans rendered as impostors. Indeed, if the normals were not saved, a terrible flat shading would be applied to the virtual human, since it is represented with only two triangles. Switching from a rigid mesh to an impostor would thus lead to awful popping artefacts. In Figure 3.3 (center), we show an example of such a representation for humans in walking postures. The generated image RGB components represent the directions of the human's normals.

- A UV map, storing in its texels the 2D texture coordinates as RG components. This information is also very important, because it allows to correctly apply a texture to each texel of an impostor. Otherwise, we would need to generate an atlas for every texture of a human template. We show an example of a UV map for a walking human in Figure 3.3 (right)

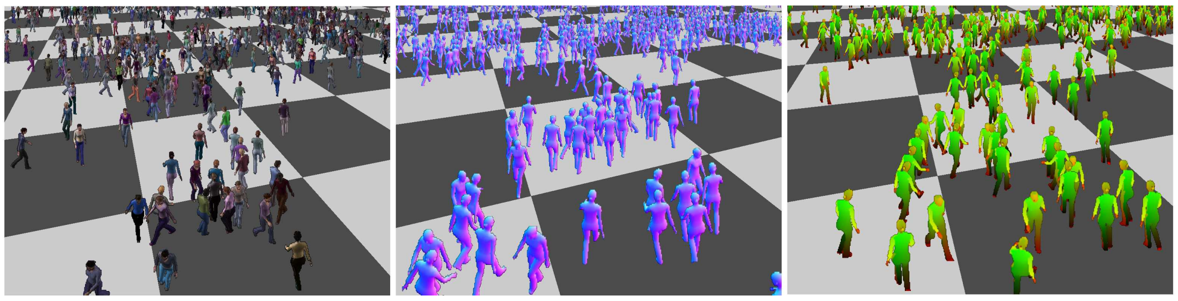


Figure 3.3: Images of walking humans. *(Left)* The textured and shaded results, as rendered in *YaQ*. *(Center)* The RGB channels of the image are used to represent the 3D directions of the normals: Red in the X direction, Green in the Y direction, and Blue in the Z direction. *(Right)* The RG channels of the image are used to represent the texture coordinates.

Since impostors are only 2D quads, we need to store normals and texture coordinates from several points of view, so that, at runtime, when the camera moves, we can display the correct keyframe from the correct camera view point.

In summary, each texture described above holds a single mesh posture for several points of view. This is why we also call such textures atlas. We illustrate in Figure 3.4 a 1024×1024 atlas for a particular keyframe. The top of the atlas is used to store the UV map, and its bottom the normal map.

The main advantage of impostors is that they are very efficient, since only two triangles per virtual human are displayed. Thus, they constitute the biggest part of the crowd. However, their rendering quality is poor, and thus they cannot be exploited close to the camera. Moreover, the storage of an impostor animation is very costly, due to the high number of textures that need to be saved.

We summarize in Table 3.1 and Figure 7.4 the performance and animation storage for each virtual human representation. We observe that each step down the representation hierarchy allows to increase by an order of magnitude the number of displayable characters. We also note that the faster the display of a representation, the larger the memory storage. Fi-

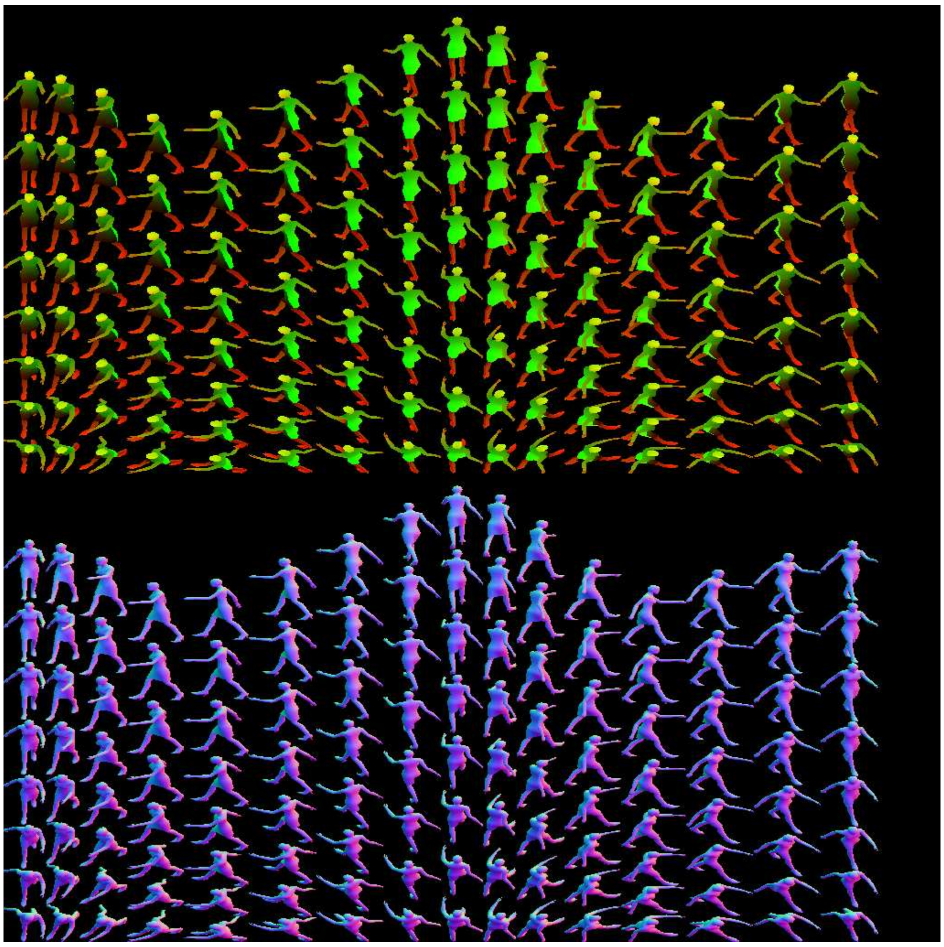


Figure 3.4: A 1024×1024 atlas storing the UV map (above) and the normal map (below) of a virtual human performing a keyframe of an animation from several points of view.

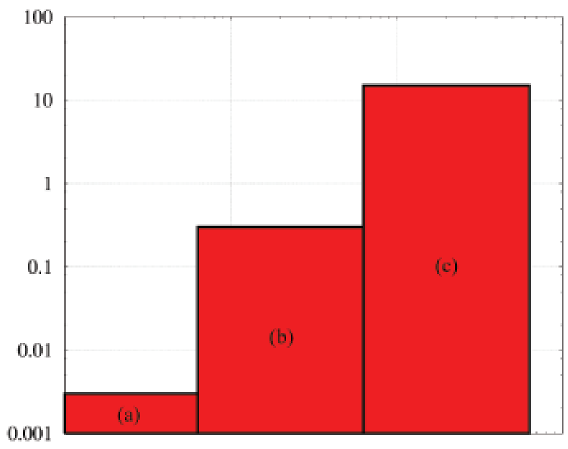


Figure 3.5: Storage space in $[Mb]$ on a logarithm scale for one second of animation of each level of detail: *(a)* deformable mesh, *(b)* rigid mesh, *(c)* impostor.

nally, rigid meshes and impostors are stored in GPU memory, which is usually much smaller

Level of detail	Max Displayable Number @$30Hz$	Animation Frequency $[Hz]$	Animation Storage Cost$[Mb/s]$	Memory Location
deformable mesh	~ 200	25	~ 0.03	CPU
rigid mesh	$\sim 2,000$	20	~ 0.3	GPU
impostor	$\sim 20,000$	10	~ 15	GPU

Table 3.1: Characteristics and costs associated to the animation of each level of detail.

than CPU memory.

3.3 *YaQ* Data

The main problem when dealing with thousands of characters is the quantity of information that needs to be stored and processed for each one of them. Processing the data is very demanding, even for modern processors, while storing it requires intelligent structures that can be efficiently accessed and shared.

In this section, we first present how *YaQ* tightly packs the data associated to a human template in order to avoid storing information that can be shared over the levels of detail, for instance. Then, we present how lists of human instances sharing similar structures can be organized to limit the number of state switches at runtime. Finally, we introduce our database, where all the heavy pre-computed data is serialized and stored to make the initialization phase of the simulation as fast as possible.

3.3.1 Human Template Data

The organization of resources inside a human template is illustrated in Figure 3.6.

Appearance Set. As previously stated, appearance sets, which we detail in Chapter 4, are used to apply various colors to the body parts of a human instance. Each appearance set is unique, and associated to a single human template. There can be several appearance sets per template, but never more than one template per appearance set. When creating an instance of a human template, one appearance set belonging to the template is chosen to modulate the colors of this instance. If a human template is instantiated several times, some instances will share the same appearance set.

Deformable Mesh. Storing a deformable mesh in *YaQ* sums up to saving four arrays: one array filled with all the vertices of the 3D mesh, one array to store the normals associated to all vertices, one array of UV coordinates, and finally, one array of indices. The indices allow to correctly draw the mesh by grouping vertices into triangles, using their corresponding normals for the shading, and mapping the texture with the UV coordinates.

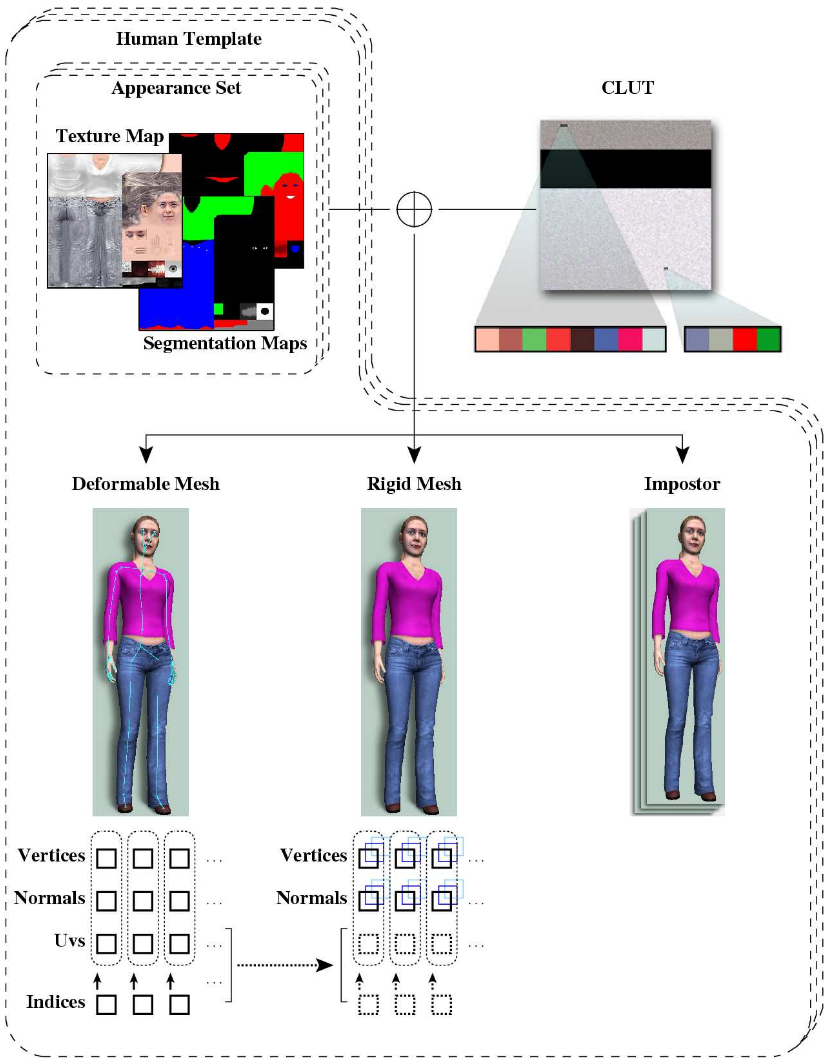


Figure 3.6: Shared resources between representations of a human template: *(a)* deformable mesh, *(b)* rigid mesh, *(c)* impostor.

Rigid Mesh. A rigid mesh is always associated to an animation, since already deformed. To store one keyframe of a rigid animation, the whole set of vertices needs to be stored again, for they are not placed at the same positions as in a deformable mesh. The same applies to the normals. The UV coordinates remain the same, because the texture is still mapped in the same way. Finally, the indices can be reused too from a deformable to a rigid mesh, if care is taken to keep the same ordering in the different arrays.

Impostors. For impostors, no information is shared with the other representations. All the data required to shade and texture the quad are stored in the atlas presented in Section 3.2.4.

3.3.2 Database Management

Many elements of data used at runtime in *YaQ* are permanent, *i.e.*, they are not changed over the whole simulation, and can even be reused between simulations. In order to keep them available at all times and to store them efficiently, we use an external database.

To animate virtual humans, the locomotion engine of Glardon *et al.* [Glardon et al., 2004a,b] is used to generate various locomotion cycles (see Chapter 6). Although this engine is fast enough to generate a walk or run cycle in real time, it cannot keep up that rhythm with thousands of virtual humans. When this problem first occured, the idea of precomputing a series of locomotion cycles and storing them in a database came up. Since then, this system has proved very useful for storing other permanent data. The main tables that can be found in our database are the following:

- Skeletal animations,
- Rigid animations,
- Impostor animations,
- Motion kits,
- Human templates, and
- Accessories.

In this section, we detail the advantages and drawbacks we meet by using such a database, and what kind of information we can safely store there.

All skeletal, rigid and impostor animations we use in real time can neither be generated online, nor at the initialization phase of the application, because the user would have to wait during an important amount of time before the simulation launches. This is why the database is used. With it, the only work that needs to be done at initialization is to load the animation sequences, so that they are ready when needed at runtime. Although this loading phase may look time consuming, it is quite fast, since all the animation data is serialized into a binary format. Within the database, the animation tables have four important fields [1]: *unique id*, *motion kit id*, *template id* and *serialized data*. For each animation entry A, its motion kit id is later used to create the necessary links (further detailed in Chapter 6), while its template id is needed to find to which human template it belongs. It also allows to restrain the number of animations to load to the strict minimum, *i.e.*, only those needed for the human templates used in the application. It is mainly the serialized data that allows to distinguish a skeletal from a rigid or a impostor animation. Note here that storing a skeletal animation is different from storing a deformable mesh, which we have previously detailed in this section. For a skeletal animation, we mainly serialize into the database all the information concerning the orientation of each skeleton joint at each keyframe. With a rigid or an impostor animation however, since the mesh or the image is already animated, the data to store is the same as previously described.

Another table in the database is used to store the motion kits. It is important to note that since they are mainly composed of simple data, like integers and strings, they are not

[1]A field can be understood as a column in the database that allows for queries.

serialized in the database. Instead, each of their elements is introduced as a specific field. When loading a motion kit M from the database, its basic information is directly extracted to be saved in our application. The motion kits are thoroughly presented in Chapter 6.

We have inserted in the database a table in order to store the permanent data of human templates. Indeed, we have some human templates already designed and ready to be used in the crowd simulation. This table has the following fields: *unique id*, *name*, *skeleton hierarchy*, and *skeleton posture*. The skeleton hierarchy is a string summarizing the skeleton features, *i.e.*, all the joint names, ids, and parent. When loading a human template, this string is used to create its skeleton hierarchy. The skeleton posture is a string giving the default posture of a skeleton: with the previous field, the joints and their parents are identified, but they are not placed. In this specific field, we get for each joint its default position and orientation, relatively to its parent.

Finally, the database possesses two tables dedicated to accessories. An accessory is a mesh used to add variety and believability to the shape of the virtual humans. For instance, it can be a wig, a pair of glasses, a bag, *etc.* (see Chapter 5 for more details). In a first table, we store the elements specific to an accessory, independently from the human template wearing it: *unique id*, *name*, *type*, *serialized data*. In the serialized data is stored all the vertex, normal and texture information to make an accessory displayable. The second table is necessary to share information between the accessories and the human templates. As later specified in Chapter 5, the displacement of a specific accessory relatively to a skeleton joint is different for each human template. This displacement is stored as a matrix. So, in this second table, we employ a field called *template id* and a field *accessory id* to know exactly where the field *matrix* must be used. Thus, for each accessory / human template couple, corresponds an entry within this table. Note that we also store there the joint to which the accessory needs to be attached. This is because in some special cases, they may differ from a skeleton to another. For instance, when we attach a back pack to a child template, the joint used is a vertebra that is different from the one for an adult template.

Using a database to store serialized information has proven to be very useful, because it greatly accelerates the initialization time of the application. The main problem is its size, which increases each time a new element is introduced into it. However, with real-time constraints, we allow ourselves to have a sufficiently large database within reasonable limits to obtain varied crowds.

CHAPTER 4

Appearance Variety

When simulating a small group of virtual humans, it is easy to make them look singularly different: one can use several human templates and textures for each virtual human present in the scene, and assign them different animations. However, when the group extends to a crowd of thousands of people, this solution becomes unfeasible. First, in terms of design, it is unimaginable to create one mesh and series of animations per individual. Moreover, the memory space required to store all the data would be far too demanding. There is no direct solution to this problem, but it is however possible to achieve good results by multiplying the levels where variety can be introduced.

First of all, the visual appearance of individuals can be varied: several human templates can be used. Then, for each template, several textures can be designed. Also, the color of each part of a texture can be varied so that two virtual humans issued from the same template and sharing the same texture have not the same clothes / skin / hair color.

Second, we can also modify the shape of human instances with several methods: we have developed the idea of accessories, *i.e.*, “augmenting” a human mesh with various objects such as a hat, a watch, a back pack, glasses, *etc*. Also, we have recently worked on modifying the shape of human instances directly. Based on a standard human template, variations are applied to the skeleton and vertices, in order to modifiy each instance’s obesity and height. This aspect of variety is presented in Chapter 5.

Finally, variety can be achieved through animation. We mainly concentrate on the locomotion domain, where we vary the movements of the virtual humans in two ways. Firstly, by generating in a preprocess several locomotion cycles (walks and runs) at different speeds, that are then played by the virtual humans online. Secondly, we use offline inverse kinematics to enhance the animation sequences with particular movements, like having a hand in the pocket, or at the ear as if making a phone call. Animation variety is presented in Chapter 6.

In the following sections, we further develop each necessary step to vary a crowd in appearance: in Section 4.1, we show the three levels where variety can be achieved. Then, in Section 4.2, we present our early work on color variety, and how we segmented the texture of a virtual human into body parts. In Section 4.3, we introduce the segmentation maps, *i.e.*, an improved approach to differentiate body parts and apply better variations in colors. Finally, in Section 4.4, we present our conclusion on the topic of crowd appearance variety.

4.1 Appearance Variety at Three Levels

When referring to appearance variety, we mean how we modulate the rendering aspect of each individual of a crowd. In the context of our wokr, this term is completely independent from the animation sequences played, the motion planning or the behavior of the virtual humans. First of all, let us remind that a human template is a data structure containing: a skeleton, defining what and where its joints are, a set of meshes, representing its different levels of detail, several appearance sets, *i.e.*, textures and their corresponding segmentation maps, and finally, a set of animation sequences that can only be played by this human template. For further indications on the human template structure, the reader is invited to refer to Section 3.2.1. We apply appearance variety at three different levels.

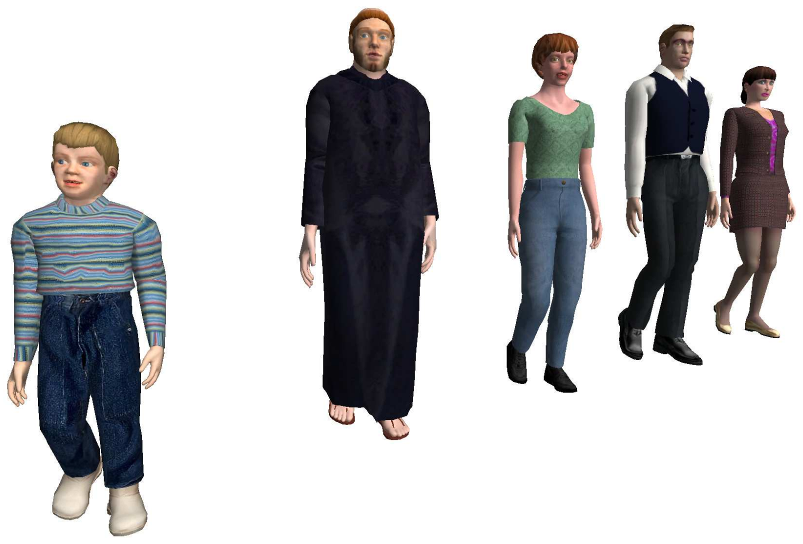


Figure 4.1: Five different human templates: their shape, textures, skeletons are different.

The first coarsest level is simply the number of human templates used. It seems obvious that the more human templates, the more variety. In Figure 4.1, we show five different human templates. The main issue when working with many human templates is the time required to design them first, and the memory requirements to store them. Their number thus needs to be limited. In order to mitigate this problem, we further vary the human templates by creating several textures and appearance sets for each one of them. An appearance set is defined as a texture and its associated segmentation maps (more on this in Section 4.3).

The second level of variety is represented by the texture of an appearance set. Indeed, once an instance of a human template is provided with an appearance set, it automatically assumes the appearance of the corresponding texture. Of course, changing appearance set, and thus, texture, does not change the shape of the human template. For instance, if its mesh contains a pony tail, it will remain whatever the texture applied. However, it can impressively modify the appearance of the human template. In Figure 4.2, we show five different textures applied to the same human template.

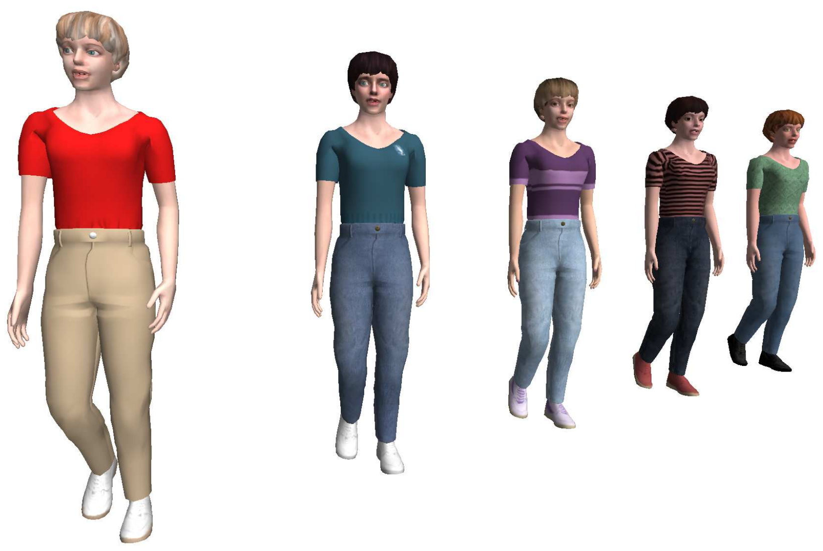


Figure 4.2: Five different textures mapped onto five instances of the same human template.

Finally, at the third level, we can play with color variety on each body part of the texture, thanks to the segmentation maps of the appearance set. In Figure 4.3, we show several instances of a same human template, using the same texture; only the body part colors are modulated. We fully dedicate Section 4.3 to this particular level. But first, and in order to fully understand segmentation maps, we briefly explain in Section 4.2 our previous approach to color variety [de Heras Ciechomski et al., 2005] and its limitations, which have been later overcome with segmentation maps.

4.2 Previous Work on Color Variety

To obtain variation inside a single texture, previous work on color variety proposed a solution to differentiate character body parts, and then apply a unique combination of colors to each of them.

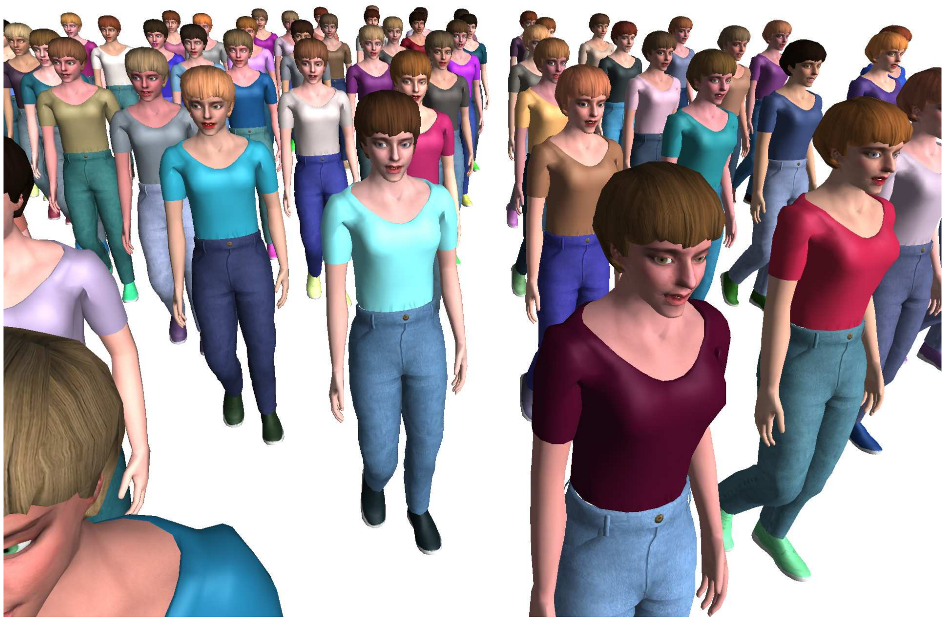


Figure 4.3: We instantiate several times the same human template with the same texture. Only per-body-part color variety is used to modify their appearance.

4.2.1 Principles

Previous work increasing variety in color appearance for the characters composing a crowd share the common idea of storing the segmentation of body parts in a single alpha layer, *i.e.*, each body part is represented by a defined level of intensity of the alpha channel. Figure 4.4 depicts a typical texture and its associated alpha zone map. The method is based on texture color modulation: the final color C_b of each body part is a modulation of its texture color C_t by a random color C_r:

$$C_b = C_t C_r. \tag{4.1}$$

Colors C_b, C_t, and C_r can take values between 0.0 and 1.0. In order to have a large panel of reachable colors, C_t should be as bright as possible, *i.e.*, near to 1.0. Indeed, if C_t is too dark, the modulation by C_r will give only darker colors. On the other hand, if C_t is a bright color, the modulation by C_r will provide not only bright colors, but also dark ones. This explains why part of the texture has to be reduced to a bright luminance, *i.e.*, the shading information and the roughness of the material. The drawback of passing the main parts of the texture to luminance is that funky colors can be generated, *i.e.*, characters are dressed in colors that do not match. Some constraints have to be added when modulating colors randomly.

4.2.2 HSB Color Spaces

Once the different body parts are identified with the alpha channel of the human texture, it is important to constrain the colors that each part can take. Indeed, if no constraint was applied at this stage, virtual humans would end up with completely random colors, as illustrated in

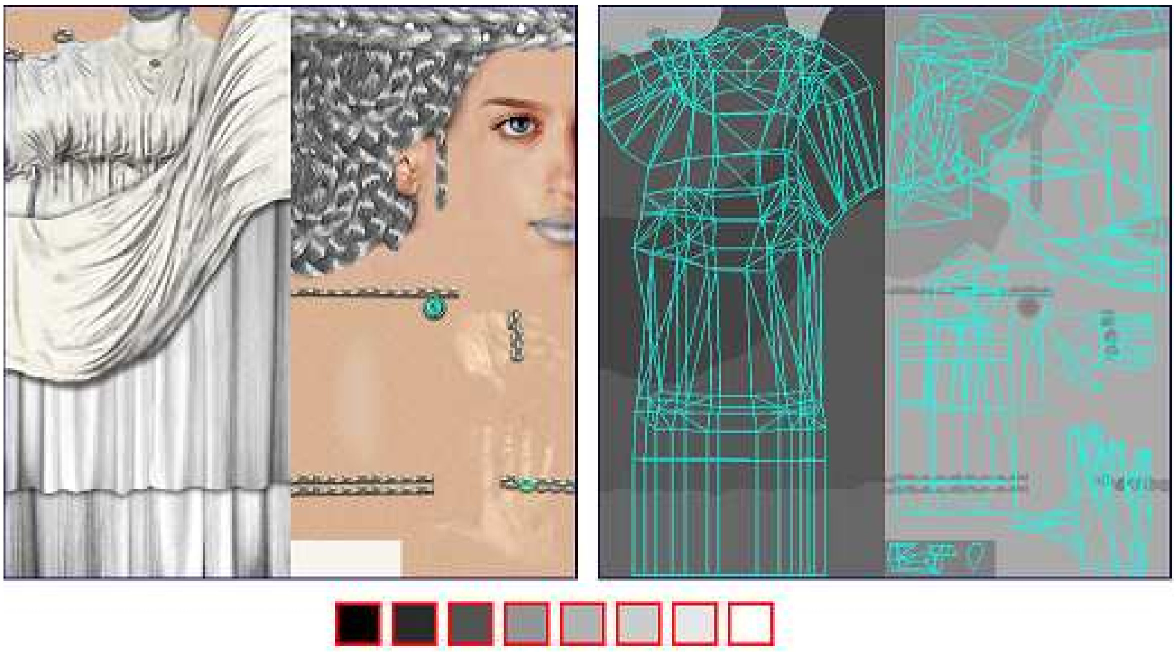


Figure 4.4: Texture used for color variety. *(Left)* the RGB channels represent the basic texture, which is very bright to allow for color modulations. *(Right)* the alpha channel of the texture is used to differentiate body parts. Each body part takes a specific alpha intensity.

Figure 4.5.

The usual RGB system represents a color with three values corresponding to the contribution of the three primary colors: red, green, and blue. With this system however, it is very counterintuitive for a designer to constrain per-body-part color ranges effectively. We have created a dedicated tool to help the designer in this task, using a different mode of color representation. [Smith, 1978] proposed a model that deals with everyday life color concepts, *i.e.*, hue, saturation and brightness, which are closer to the human color perception than the RGB system. This system is called the HSB (or HSV) color model (see Figure 4.6):

- the hue defines the specific shade of color, as a value between 0 and 360 degrees;
- the saturation denotes the purity of the color, *i.e.*, highly saturated colors are very bright, while low saturated colors are washed-out, like pastels. Saturation can take values between 0 and 100,
- the brightness measures how light or dark a color is, as a value between 0 and 100;

In the process of designing virtual human color variety, localized constraints are dealt with: some body parts need very specific colors. For instance, skin colors are taken from a specific range of unsaturated shades with red and yellow dominance, almost deprived of blue and green. Eyes are described as a range from brown to green and blue with different levels of brightness. These simple examples show that one cannot use a random color generator as is. The HSB color model offers control on color variety in an intuitive an flexible manner. Indeed, as shown in Figure 4.7, by specifying a range for each of the three parameters, it is possible to define a 3D color space, called the HSB map.

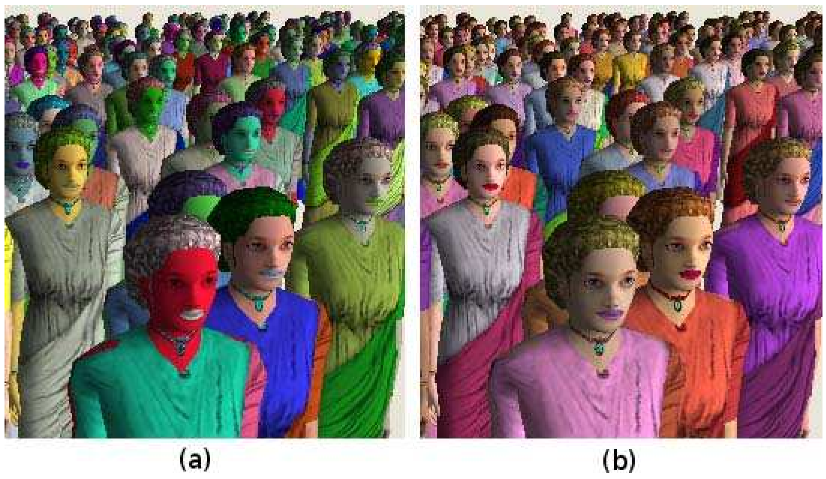


Figure 4.5: *(a)* Human instances have their body part colors randomly chosen. *(b)* For each body part, a designer has constrained the range of possible colors using our dedicated tool, based on the HSB color model.

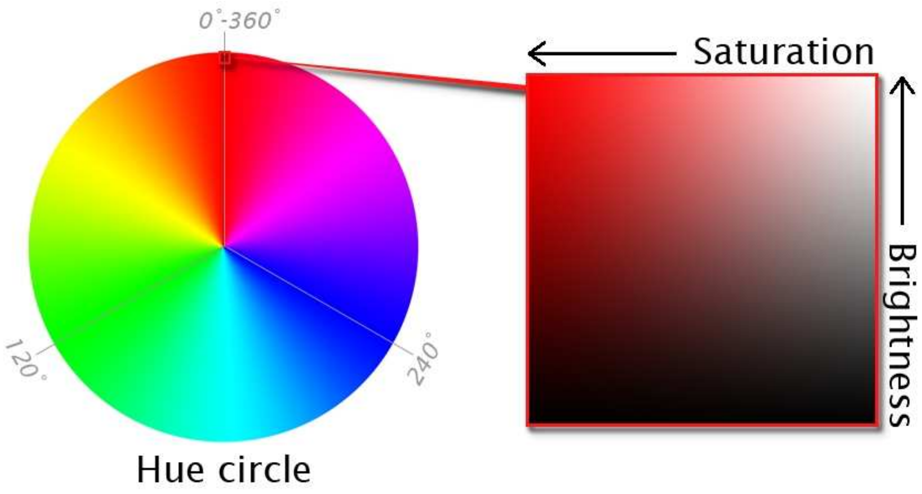


Figure 4.6: HSB color space. Hue is represented by a circular region. A separate square region may be used to represent saturation and brightness, *i.e.*, the vertical axis of the square indicates brightness, while the horizontal axis corresponds to saturation.

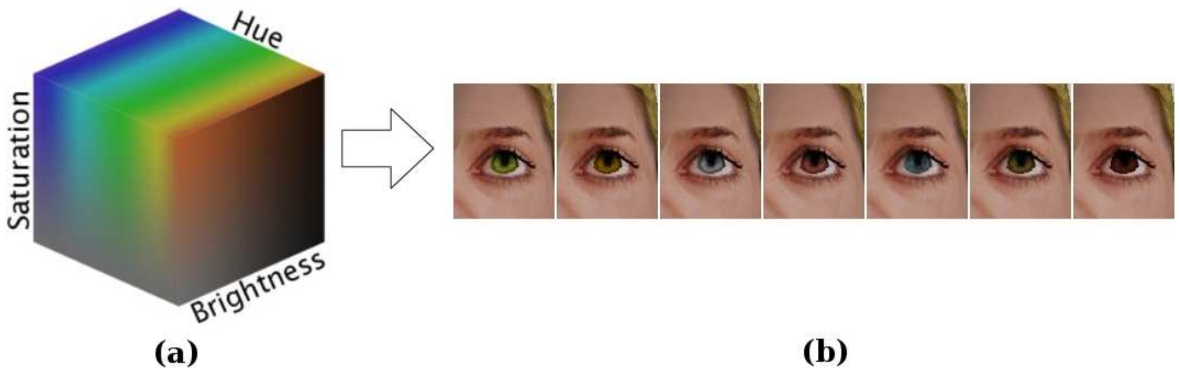


Figure 4.7: The HSB space is constrained to a three dimensional color space with the following parameters *(a)* hue from 20 to 250, saturation from 30 to 80 and brightness from 40 to 100. *(b)* Colors are then randomly chosen inside this space to add variety on the eye texture of a character.

4.3 Segmentation Maps

To further vary the instances of a same human template, the method presented in Section 4.2 is perfectly adequate when viewing crowds at far distances. However, using a single alpha layer to segment body parts has several drawbacks. No bilinear filtering can be used on the texture, because incorrect interpolated values would be fetched in the alpha channel at body part borders, as shown in Figure 4.8 (a). Moreover, for individuals close to the camera, the method tends to produce too sharp transitions between body parts, *e.g.*, between skin and hair, as depicted in Figure 4.8 (b), due to the impossibility of associating a texel to several body parts at the same time. Also, character close-ups bring the need for a new method capable of handling detailed color variety. Subtle make-up, or detailed patterns on clothes greatly increase the variety of a single human template. Furthermore, changing illumination parameters of materials, *e.g.*, their specularity, provides more realistic results. Previous methods would require costly fragment shader branching to achieve such effects. We apply a versatile solution based on *segmentation maps* to overcome previous method drawbacks.

4.3.1 Principles

For each texture of a human template, we create a series of segmentation maps. A segmentation map is a four channel image (RGBA), delimiting four body parts *i.e.*, one per channel, and sharing the same parameterization as the texture of the appearance set (compare the right-side of Image (b) with the left-side of Image (d) in Figure 4.8). This method allows for each texel to partially belong to several body parts at the same time through its channel intensities: the intensity of each body part is defined throughout the whole body of each character, *i.e.*, 256 levels of intensity are possible for each part, 0 meaning it is not present at this location, and 255 meaning it is fully present. As a result, it is possible to design transitions between body parts much smoother than in previous approaches, as shown on the close-up of Figure 4.8 (c) and (d). Moreover, using segmentation maps to efficiently distinguish body parts provides two additional advantages over previous methods:

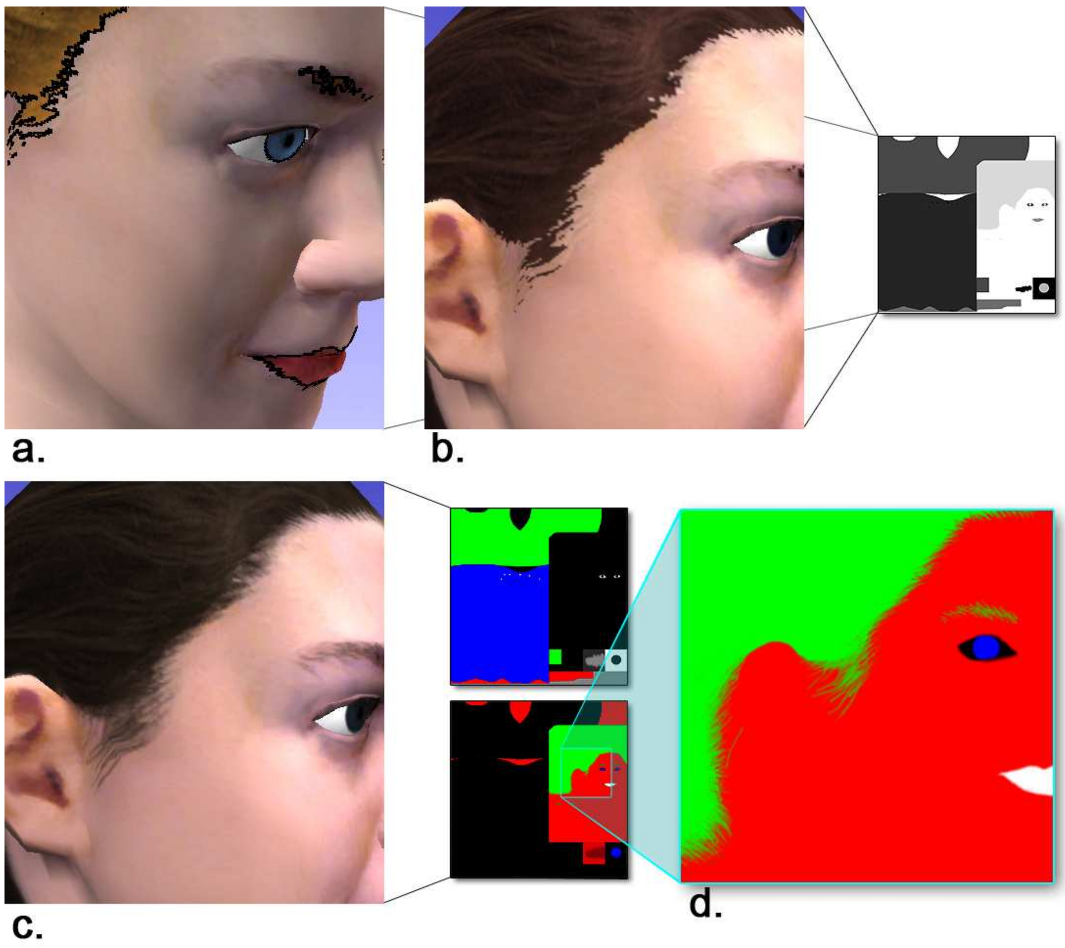


Figure 4.8: Close-up on the transition between skin and hair: artifacts in previous methods *(a)* with bilinear filtering and *(b)* nearest filtering. *(c)* Smooth transitions obtained with segmentation maps and bilinear filtering. *(d)* Detailed transition between skin (red channel) and hair (green channel) on the segmentation map.

- Possibility to apply different illumination models to each body part. With previous methods, achieving such effects requires costly fragment shader branching.
- Possible mipmapping activation and use of linear filtering, which greatly reduce aliasing. Since previous methods use the alpha channel of the texture to segment their body parts, they cannot benefit from this algorithm, which causes the appearance of artefacts at body part seams (see Figure 4.8).

We have empirically determined to use eight body parts, *i.e.*, two RGBA segmentation maps for each appearance set. The results obtained with eight body parts are satisfying for our specific needs, but the method can be used with more segmentation maps if more parts are needed. For instance, it would be possible to use the method for adding color variety to a city by creating segmentation maps for buildings, as introduced in Section 8.1.

To provide one more level of variety, it is possible to define several pairs of segmentation maps per human template texture, allowing to create different patterns as illustrated in

Figure 4.9: make-up, cloth patterns, freckles, *etc.*, and localized specular parameters.

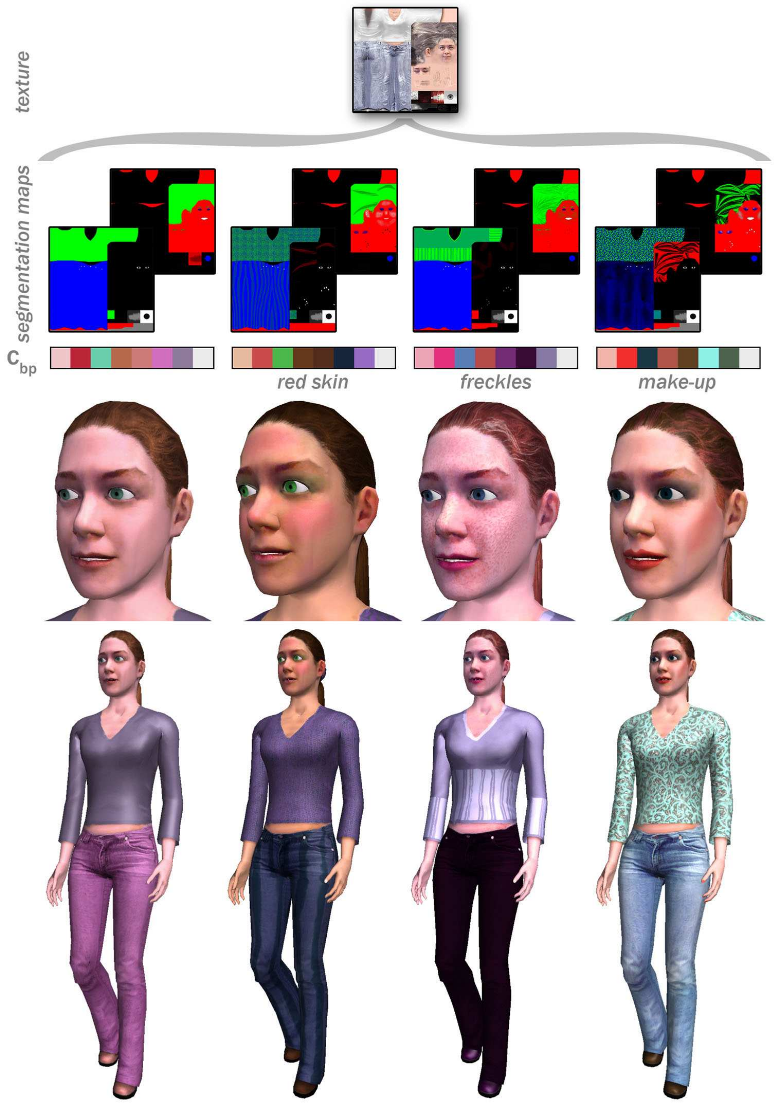


Figure 4.9: *(top to bottom)* a human template original texture; several pairs of segmentation maps sharing the same parameterization; examples of unique sets of eight colors for each character; detailed effects obtained with segmentation maps and specularity parameters on faces (make-up, freckles, glossy lips, etc.) and on the whole body (cloth patterns, shiny shoes, etc.).

Ideally, for a given pixel of the texture, we wish the sum of the intensities of each body part to reach 255, *i.e.*, a texel partially belongs to several body parts, but the sum of these adherences should reach 100%. However, segmentation maps are designed manually, with a software like Adobe Photoshop [Adobe, 2009], and it may happen that the sum of intensity levels for some texels do not reach 255. In this case, unwanted artefacts may later appear within the smooth transitions between body parts. For instance, imagine the transition between the hair and the skin of a virtual human. A pixel of the segmentation map may reach a contribution of 100 for the skin part, while the hair part contribution is of 120. Their sum amounts to 220. Although this is not an issue while designing the segmented body parts in

Photoshop, it leads to problems when trying to normalize the contributions in the application. Indeed, with simple normalization, such pixels compensate the uncomplete sum with a black contribution, thus producing a final color much darker than expected. This is illustrated in Figure 4.10. The proposed solution is to compensate this lack with white instead of black, to get a real smooth transition without unwanted dark zones.

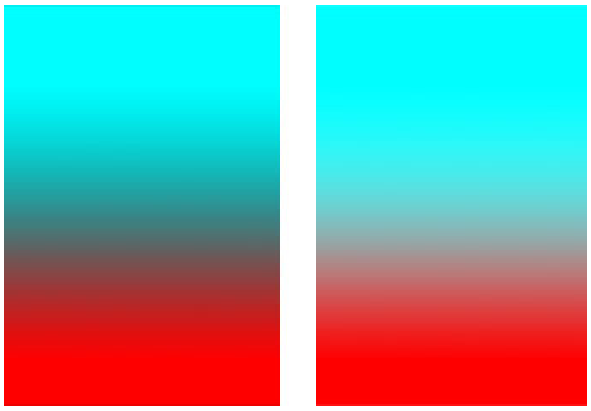


Figure 4.10: A blue to red gradient. *(Left)* The sum of the red and blue contributions does not reach 255 in some pixels, causing the gradient to suffer from an unwanted black contribution, *(Right)* A white contribution is added so that the sum of contributions is always 255.

4.3.2 Color Computation and Storage

At its creation, a character is assigned a unique set of eight RGB colors c_{bp} for its eight body parts (also illustrated in Figure 4.9), randomly selected within constrained color spaces. For each pixel composing a body, a final color is computed as a combination of these eight colors, weighted by the channel intensities of the segmentation maps. For instance, a body pixel p with texture coordinates (u, v) has its final color c_p computed as:

$$c_p = c_t \sum_{s \in (S_1, S_2)} \sum_{a \in (R,G,B,A)} I_{s,a}(u, v) c_{bp}(s, a). \tag{4.2}$$

Here, c_t is the color of the original texture (top of Figure 4.9) at coordinates (u, v). The identifier s represents either the first or the second segmentation map from the pair used: (S_1, S_2). The function defined as $I_{s,a}(u, v)$ is the intensity of the texel with coordinates (u, v) for channel a of segmentation map s, and finally, $c_{bp}(s, a)$ is the color of the corresponding body part. Note that Equation 4.2 computes a sum on two segmentation maps and four channels per segmentation map, totaling to eight components for eight body parts.

To compute the final pixel color of each character, we have implemented a dedicated fragment shader (see Section 10.2 of the *Appendix*). At its creation, each character is assigned a unique set of eight random colors c_{bp} to be applied to the eight body parts. Sending these colors to the shader as uniforms would be time consuming. Instead, eight unique body part colors per character are computed at initialization, and stored in eight contiguous RGB texels, of a 1024×1024 image, called Color Look-Up Table (CLUT). Once filled, the CLUT is only sent once to the shader. In order to arrange as many sets of eight RGB colors as possible, the alpha channel of the look-up image is also exploited. Thus, only six RGBA texels per virtual human are used.

Therefore, it is possible to store a set of up to: $\frac{1024 \times 1024}{8} = 131,072$ unique combinations of colors. Previous methods were limited to $4,096$ combinations, because they could not address every row of the look-up image by only using the alpha channel of a human texture. Finally, to further improve detailed variety, we also assign each body part specularity parameters. Note that these parameters are not saved within the CLUT, but directly sent to the GPU. We show an illustration of a CLUT in Figure 4.11.

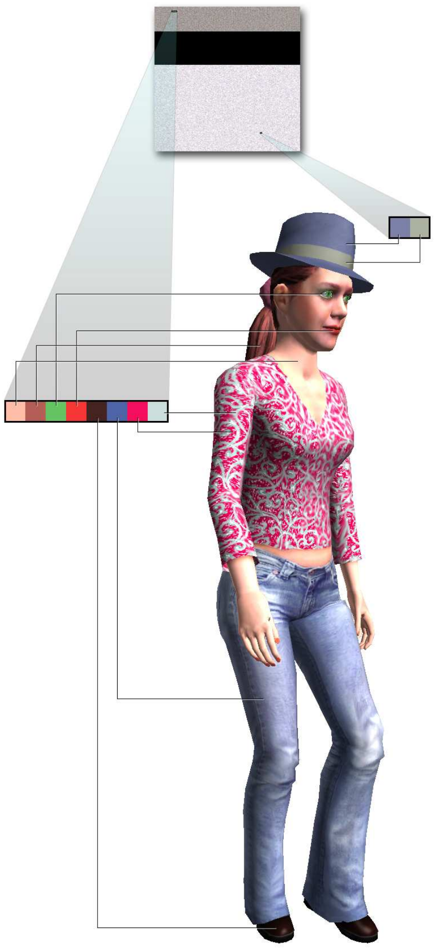


Figure 4.11: A CLUT image used to store the color of each virtual human body parts and accessories (detailed in Section 4.3.2).

In terms of storage, each segmentation map pair corresponds to two 1024×1024 textures of about 1MB each, compressed in DDS format (DXTC5). The number of pairs defined for each human template texture is up to the designer. As for the CLUT, it is not compressed to avoid artifacts when reading pixel by pixel the colors to apply to the characters. It approximately weights 4MB. The total memory storage cost for using our approach is thus only dependent on the number of human templates exploited and the appearance sets defined for

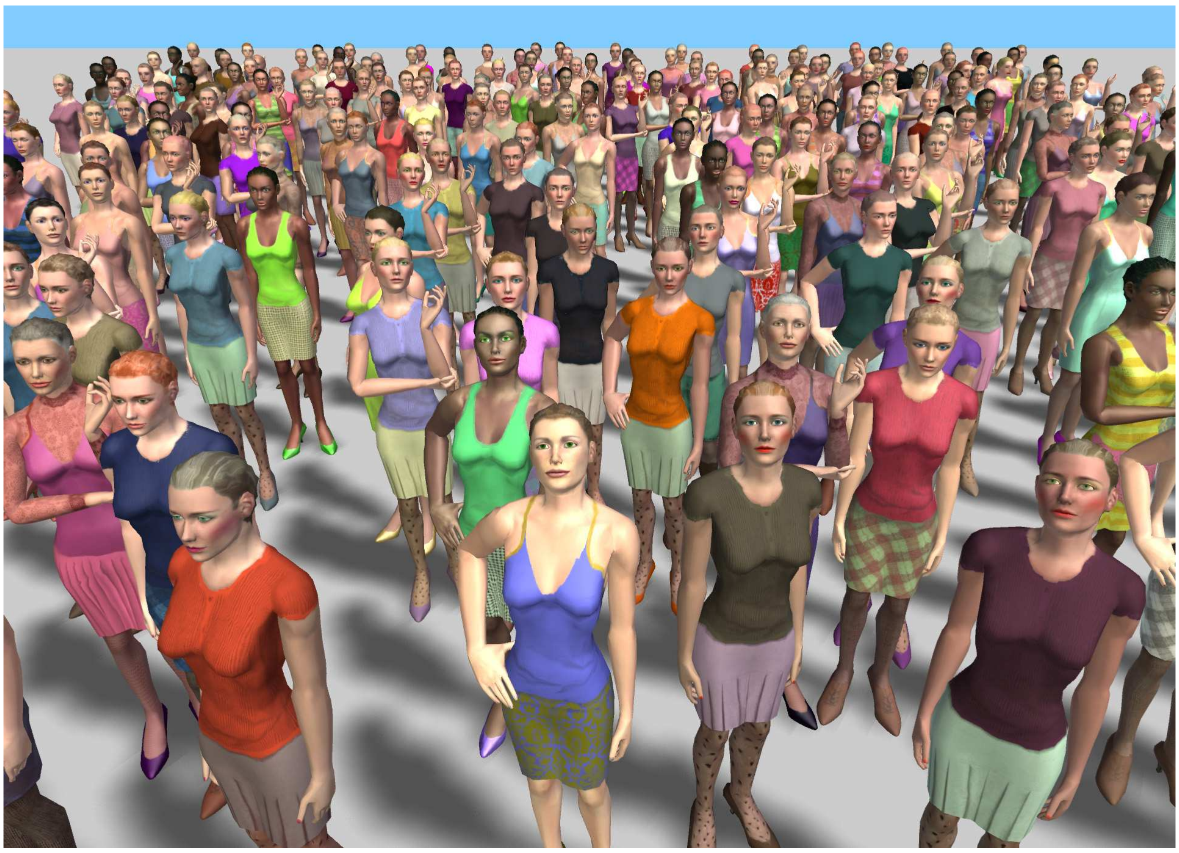


Figure 4.12: Appearance sets applied to instances of a single human template. Note the different specular effects on the body parts and the varying cloth patterns.

each of them. Segmentation maps are generic: they can be used in any number and on any textured object to vary its appearance. For instance, to segment instances of accessories (see Figure 1.2). They are also scalable to any LOD, and thus, keep the appearance of a character consistent. In Figure 2.1, we illustrate this scalability on a single character rendered in three LOD. The final results obtained by introducing appearance variety are illustrated in Figure 4.12, where several instances of a single human template are displayed, taking full advantage of all available appearance sets and color variety.

We detail in Section 10.2 of the *Appendix* one approach to implement segmentation maps in a fragment shader, using GLSL with Shader Model 3.0 hardware.

4.4 Conclusion

In this chapter, we have introduced a new simple technique to apply variety in visual appearance to object instances. The first step to add visual appearance variety is to create several different textures for each template. The next step is to vary the colors of clothes, skin and hair of instances sharing the same texture. For that, we complement the usual template texture with two segmentation maps that allow delimiting eight body parts that can be colorized differently for each instance at runtime. Finally, in order to simulate different types and colors of materials, the designer can specify illumination parameters for each body part.

Thanks to this technique, crowds are enhanced with subtle make-up, freckles and beard effects, or detailed cloth patterns, and smooth transitions between body parts are ensured (see Figure 3.2). At initialization, the instances are created, and random colors within defined spaces are chosen for each of them. The sets of chosen colors are then contiguously stored in a Color Look-Up Table on the GPU.

All the successive steps of this technique have been fully detailed, and a commented fragment shader, implemented in GLSL, can be found in Appendix 10.2. We have illustrated the use of appearance sets on several examples, demonstrating their versatility. Moreover, appearance sets are easily scalable for all LOD commonly exploited.

CHAPTER 5

Shape Variety

We have already described in Chapter 4 how to obtain varied clothes and skin colors by using several appearance sets. Unfortunately, even with these techniques, the feeling of watching the same person is not completely overcome. The main reason is the lack of variety in the human templates used. Indeed, it is very often the same human template (or a small number of them) that is used for the whole crowd, resulting in large groups of similarly shaped humans. We cannot increase too much the number of human templates, because it requires a lot of work for a designer to create the human template, its textures, its skinning, its different levels of detail, *etc.* Note that the number of human templates is also limited by the storage capacity of the computer running the simulation. To further add variety to characters composing the crowd, it is possible to modify their shape. In this chapter, we present two methods to achieve this. First of all, characters can be accessorized with items such as bags, hats, wigs, glasses, moustaches, *etc.* Secondly, we propose a technique to modify the height of a human skeleton, and the shape of its mesh.

Accessorizing crowds is kept simple and efficient by writing down two assumptions:

(1) Accessories are not deformed. This makes the designing phase much simpler, and also alleviates the underlying runtime computations.

(2) An accessory is associated to a human template by attaching its vertices to a single specific joint of the character's skeleton. Thus, every movement of this single joint directly reflects on the attached accessory.

Although these assumptions limit the variety of accessories, they greatly simplify their creation and usage. Also, the available set of accessories is still large enough to offer a great improvement on crowd variety, as illustrated in Figure 3.2.

To add variety to the morphology of human templates, we can modify their height and shape. For each template, the designer can specify skeleton and skinning alterations. Skeleton modifications allow for global height variations and more localized effects such as broad or narrow shoulders, while skinning deformations increase the shape uniqueness by scaling each vertex independently to obtain thin, fat, muscular and pregnant templates.

We first present a general introduction to accessories and separate them into two types in Section 5.1. Then, in Section 5.2, we show how it is possible to adapt them for all rendering levels of detail (LOD), *i.e.*, deformable meshes, rigid meshes, and impostors. In Section 5.3, we describe our implementation of accessories in *YaQ*. A discussion on their limitations and possible extensions is presented in Section 5.4. Finally, we introduce how to further modify the shape and height of a human template in Section 5.5

5.1 Accessories

In real life, people have different haircuts, they wear hats or glasses, carry bags, or suitcases, *etc*. These particularities may look like details, but it is with the sum of those details that we are able to distinguish anyone. In this section, we first explain what exactly are accessories. Then, we show from a technical point of view the different kinds of accessories we have identified, and how to apply them to all rendering levels of detail.

An accessory is a simple mesh representing any element that can be added to the original mesh of a virtual human. It can be a hat as well as a handbag, or glasses, a clown nose, a wig, an umbrella, a cellphone, *etc*. Accessories have two main purposes: first, they allow to easily add shape variety to virtual humans. Second, they make characters look more believable: even without intelligent behavior, a virtual human walking around with a shopping bag or a cellphone looks more realistic than one just walking around. The addition of accessories allows a spectator to identify himself to a virtual human, because it performs actions that the spectator himself does everyday. We basically distinguish two different kinds of accessories that are incrementally complex to develop. The first group is composed of accessories that do not influence the movements of a virtual human. For instance, whether someone wears a hat or not will not influence the way he walks. The second group gathers the accessories requiring a small variation in the animation clip played, *e.g.*, a virtual human moving with an umbrella or with a bag still walks the same way, but the arm in contact with the accessory needs an adapted animation sequence.

5.1.1 Simple Accessories

The first group of accessories does not necessitate any particular modification of the animation clips played. They simply need to be correctly “placed” on a virtual human. Each accessory can be represented as a simple mesh, independent from any virtual human. First, let us lay the problem for a single character. The issue is to render the accessory at the correct position and orientation, accordingly to the movements of the character. To achieve this, we can “attach” the accessory to a specific joint of the virtual human. Let us take a real example to illustrate our idea: imagine a walking person wearing a hat. Supposing that the hat has the correct size and does not slide, it basically has the same movement as the head of the person as he walks. Technically, this means that the series of matrices representing the head movement are the same for the hat movement. However, the hat is not placed at the exact position of the head. It usually is on top of the head and can be oriented in different ways, as shown in Figure 5.1. Thus, we also need the correct displacement between the head joint position and the ideal hat position on top of it. In summary, to create a simple accessory, our needs are the following:

- For each accessory:
  - A mesh (vertices, normals, texture coordinates),
  - A texture.
- For each human template / accessory couple:
  - The joint to which the accessory must be attached,
  - A matrix representing the displacement of the accessory, relatively to the joint.

Note that the matrix representing the displacement of the accessory is not only specific to one accessory, but specific to each human template / accessory couple. This allows us to vary the position, the size, and the orientation of the hat depending on which virtual human mesh we are working with. This is depicted in Figure 5.1, where the same hat is worn differently by two human templates. It is also important to note that the joint to which the accessory is attached is also dependent on the human template. This was not the case at first: a single joint was specified for each accessory, independently from the human templates. However, we have noticed that depending on the size of a virtual human, some accessories may have to be attached to different joints. For instance, a backpack is not attached to the same vertebra if it is for a child or a grown up template. Finally, with this information, we are able to assign each human template a different set of accessories, greatly increasing the feeling of variety.

5.1.2 Complex Accessories

The second group of accessories we have identified is the one that requires slight modifications of the animation sequences played, *e.g.*, the hand close to the ear to make a phone call, or a hindered arm sway due to carrying a heavy bag. Concerning the rendering of the accessory, we still keep the idea of attaching it to a specific joint of the virtual human. The additional difficulty is the modification of the animation clips to make the action realistic. We only focus on locomotion animation sequences. Our raw material is a database of motion captured walk and run cycles that can be applied to virtual humans. There are two options to modify an animation related to an accessory. Let us take two examples, illustrating these cases:

- If we want a virtual human to carry a bag for instance, the animation modifications are limited to the arm sway, and maybe a slight bend of the spine to counterweight the bag. Such modifications can be applied procedurally, and at runtime, by blocking some joint movements, and / or clamping their rotation.

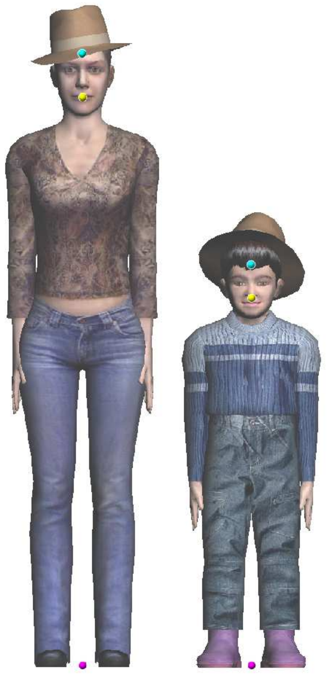


Figure 5.1: Two human templates wearing the same hat, in their default posture. The pink, yellow and blue points represent the position and orientation of the root, the head joint $(m1)$, and the hat accessory $(m2)$, respectively.

- If it is a cellphone accessory that we want to add, we need to keep the hand of the character close to its ear and avoid any collision over the whole locomotion cycle. This kind of modifications is too complex to be achieved at runtime. In such cases, we work with an inverse kinematic tool to modify the animation cycles in a pre-process. From each animation clip, an adjustment of the arm motion is performed in order to obtain a new animation clip integrating the desired movement. These animation modifications can be generalized to other movements that are independent from any accessory, for instance, hands in the pockets.

The process to render complex accessories is exactly the same as rendering simple accessories, and detailed in Section 5.3.2. The animation stage however, has some particularities for complex accessories, and requires special care. We dedicate Chapter 6 to these animation modifications.

5.2 Levels of Detail for Accessories

There are several important steps in the pre-process of modeling an accessory, so that it can later be correctly placed and oriented for all human templates. First of all, we identify the joint to which the accessory should be attached. In most cases, the same joint is selected for all human templates, but when they are too different in size, the best adapted joint can differ, *e.g.*, a backpack would be attached to a different vertebra on a child and on an adult template.

The chosen joint is called the *attach joint*. In a second phase, the accessory is transformed, so that it perfectly coincides with the mesh of each human template. These changes are expressed relatively to the attach joint as a 4×4 transformation matrix $T_{accessory}$, saved for each human template and accessory combination.

At the initialization of the crowd simulation, all characters are assigned various accessories that are then displayed at runtime. Note that these assignments are not randomly achieved. First of all, each accessory is categorized with a specific type (backpack, hat, glasses, *etc.*) and a specific theme (casual, old-fashioned, funny, *etc.*). Secondly, the human templates for which this accessory will be available are chosen by the designer. This classification allows to choose which accessories are to be used in an application, thus offering a large variety of possibilities within the limits of extravagance set by the designer. The final 4×4 transformation matrix T of the accessory in world space is computed at each time step with the equation:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = T_{char} T_{joint} T_{accessory} \tag{5.1}$$

where T_{char} is the character transformation matrix in world coordinates, and T_{joint} is the attach joint deformation matrix, relative to T_{char}.

Accessories are scalable to all LOD commonly used in crowd simulations. For deformable meshes, Equation 5.1 is directly computed at runtime. In the case of rigid meshes and impostors, we explain below dedicated methods to easily place accessories. Beforehand, it is important to mention that switching from an accessorized deformable mesh to a rigid one is unnoticeable as they share the exact same appearance. To ensure transparent switching between accessorized rigid meshes and impostors, we use the pixel to texel ratio metrics described in [Dobbyn et al., 2005].

5.2.1 Rigid Meshes

Rigid meshes are precomputed postures, or keyframes, of deformable meshes performing a given animation sequence. Their main advantage is that no dynamic deformation of the skeleton is necessary at runtime. However, it also implies that matrix T_{joint} in Equation 5.1 is not available to place an accessory.

A naive approach would be to store the accessory animation as done for a rigid mesh: store the positions of all the accessory vertices at each keyframe. However, we here take advantage from our assumption (1): the accessory's mesh is never deformed: every time a rigid mesh keyframe is stored, the matrices T_{joint} to which accessories are attached are also saved. We have identified six potentially "attachable" joints: the skull, for glasses and hats, a vertebra for backpacks, elbows, for watches and bracelets, and finally wrists, to place objects in the character's hands, given there is no finger animation. Thus, for one rigid mesh keyframe, only six matrices representing T_{joint} are saved. The storage cost for a single keyframe is thus independent from the number of accessories, whereas the naive approach would require to save one keyframe for each of them.

5.2.2 Impostors

Accessories for impostors are generated in a pre-process: image tiles are sampled all around the object in orthographic mode and saved in 512×512 normal and uv maps. In Figure 5.2 (b) and (c), we illustrate a uv map and a normal map for a hat. Sampling the circumscribed sphere of an object is often achieved with spherical coordinates, although it leads to an excessive amount of samples near the poles, as compared to the equator. Also, finding the correct tile at runtime requires costly trigonometric computations. As depicted in Figure 5.2 (a), we apply a Sukharev grid on each spherical face of a cube [Yershova and LaValle, 2004]: this method uses a cube map to discretize a sphere and distribute samples on it. With this method, samples are distributed on the sphere in a much more uniform way than with spherical coordinates. Moreover, with this method, finding the correct tile online sums up to a fast cube map look-up [Greene, 1986]. Note that the memory storage cost for an accessory impostor is constant and independent from the number of keyframes generated for a human impostor. More precisely, we store one uv map and one normal map per accessory in DDS format (DXTC5), for a total of only 680KB (including mipmaps). We note that some channels have more precision than others. Thus, to make sure the compressed atlas have as little artifacts as possible, we use the most accurate channels to store the data we need.

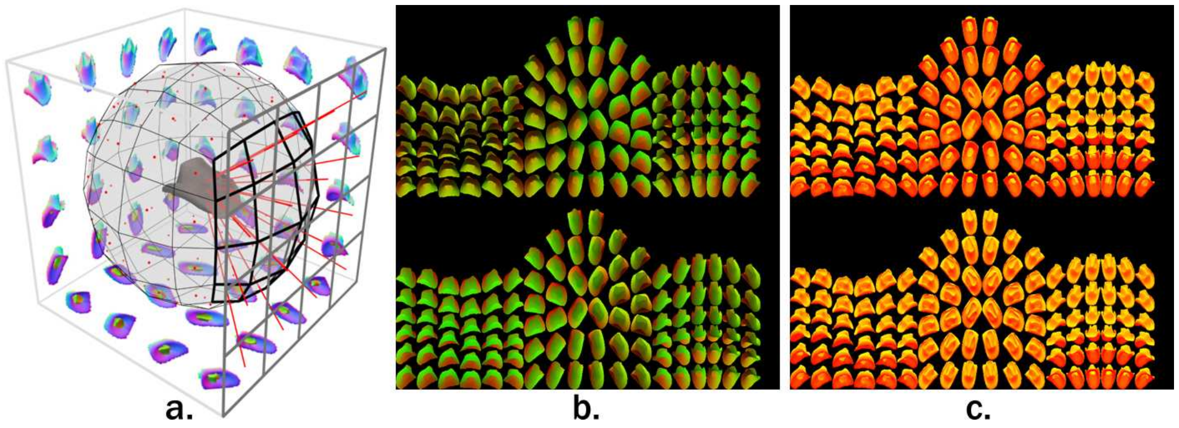


Figure 5.2: *(a)* Sampled tiles for a hat impostor creation. *(b)* The resulting uv map and *(c)* normal map.

To correctly place an accessory impostor, we use a dedicated runtime pipeline, illustrated in Figure 5.3 (a), and composed of five steps:

Step 1. Compute the accessory transformation matrix T with Equation 5.1, that provides the orientation and position of the accessory, should it be rendered as a 3D mesh. To obtain T, we pre-compute and save the six matrices T_{joint} for each human impostor keyframe, similarly to the rigid mesh approach, since no skeletal deformation is performed online for impostors either.

Step 2. Retrieve a normal and a uv map tile, representing the accessory from the correct point of view. The adequate view is deduced by expressing the current camera position relatively to T, *i.e.*, in accessory space. The resulting direction vector determines the tile to use in the cube map, as illustrated in Figure 5.3 (a) [Greene, 1986].

Step 3: Compute the exact position of the accessory impostor, *i.e.*, the quad. This position does not correspond to the one where the 3D mesh would be rendered, because the center of the quad and the center of the 3D mesh do not necessarily correspond. To have an impostor correctly positioned at runtime, we make sure, in a pre-process, and when generating each tile, to save the offset (X, Y) between the 2D quad center and the 3D accessory center, projected onto the quad. At runtime, the translation part t of T is transformed into camera space, and offset by (X, Y) so that the impostor is properly placed. Note that t is only offset on the X and Y axes in camera space. At this stage, the depth Z where the quad has to be rendered is still the same as the one originally computed in T. The algorithm to compute each fragment depth is detailed in Step 5.

Step 4: Rotate the quad around the camera Z axis so that it is correctly oriented. We illustrate the necessity of this stage in Figure 5.3 (c), where a side view of a hat positioned on a human impostor is shown. Since the character wears the hat inclined backwards, the quad has to be rotated to correctly imitate the 3D hat orientation. The computation of this rotation is achieved at runtime by first transforming the rotation part R of T into camera space, and then extracting its Z axis component.

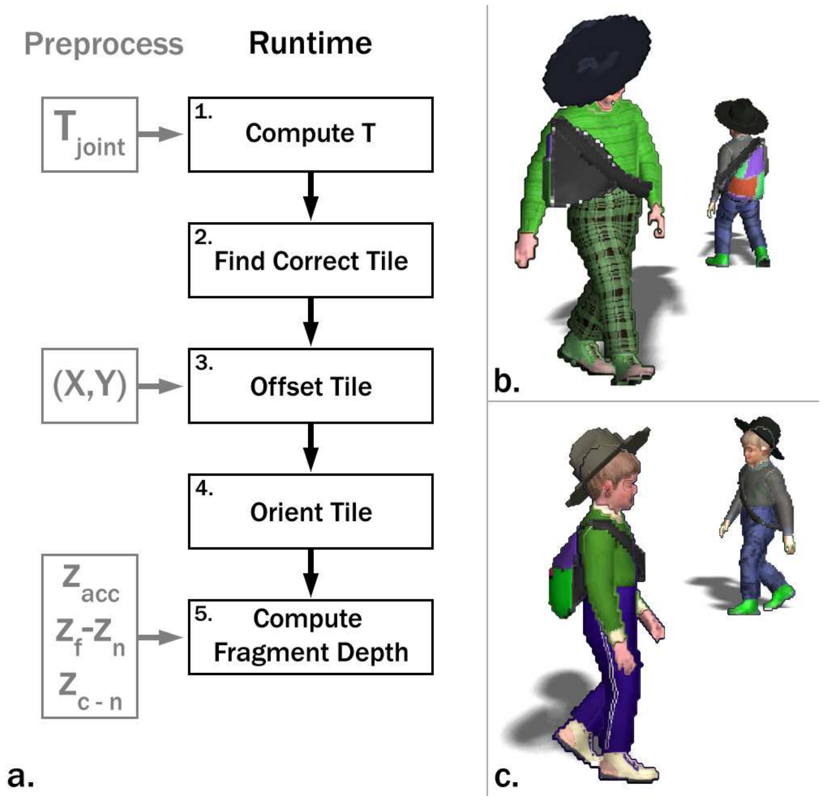


Figure 5.3: *(a)* Pipeline of accessory impostor positioning. *(b)* Impostors with no occlusion treatment. *(c)* Impostors with our fragment depth computation.

Step 5: Compute the depth of each impostor pixel (or fragment) to avoid visual artifacts, as in Figure 5.3 (b). The main problem when addressing such an issue with a perspective projection is that depth values in the Z buffer do not vary linearly. Our dedicated algorithm solves visibility problems inherent in impostors. Moreover, it is robust and allows to use any projection, even perspective. Here, we first explain the important values to compute in a preprocess, and how they are later exploited in real time to determine the depth of each fragment.

When each tile is generated offline, the near and far clipping planes are set as close to the object as possible. Note that the near plane thus has the same depth as the accessory vertex closest to the camera. Then, on the GPU, the 3D accessory vertex depth values in camera space are passed from the vertex to the fragment shader. For each fragment, these values are interpolated. We then compute, for each fragment, a final depth z_{acc} as the normalized distance between the fragment and the near plane:

$$z_{acc} = \frac{z_{frag} - z_n}{z_f - z_n} \tag{5.2}$$

where z_n and z_f are the near and far clipping plane depths, and z_{frag} is the fragment depth value interpolated from the vertices. For each fragment, z_{acc} is saved in one of the unused channels of the uv map (see Figure 5.2 (b)).

In addition, for each tile, two important parameters need be saved at this stage, still in camera space: the distance $z_f - z_n$ between the near and far planes, which later allows to denormalize z_{acc}, and the distance z_{c-n} between the depth of the 3D accessory center and the near plane.

At Step 5 of the runtime pipeline, the four quad vertices are sent untouched to the vertex shader, where they are transformed into camera space. At this moment, these vertices all have the same depth as the 3D accessory center, defined in T. They are then sent to the fragment shader, where we compute the final depth of each fragment in two operations. First, the value z_{acc} is retrieved from the uv map to be denormalized. Then, the depth z_{camera} of the fragment is computed in camera space:

$$z_{camera} = z_{frag} + z_{c-n} - (z_{acc}(z_f - z_n)) \tag{5.3}$$

where z_{frag} is the fragment depth value in camera space, interpolated from the depth of the quad vertices in the GPU. The first part of Equation 5.3 shows that each fragment has a depth of $z_{frag} + z_{c-n}$ at the *nearest*. If the accessory was to be rendered with its 3D mesh, this value would correspond to the depth of its vertex closest to the camera. From this distance, the second part of Equation 5.3 offsets each fragment by the denormalized value of z_{acc}. Based on its initial computation (see Equation 5.2), this offset is at the maximum equal to the distance between z_n and z_f, *i.e.*, the 3D accessory vertex that is the farthest from the camera.

Finally, we use the perspective transform to express z_{camera} in the canonical view volume, *i.e.*, between -1.0 and 1.0:

$$depth_{frag} = \frac{s_f + s_n}{s_{f-n}} + \frac{2.0 s_f s_n}{s_{f-n} z_{camera}} \tag{5.4}$$

where s_n and s_f are the depths of the scene near and far planes in camera space, and s_{f-n} is the distance between them. Note that in the OpenGL Shading Language, the depth assigned to each fragment ranges between 0.0 and 1.0. The result to Equation 5.4 therefore has to be normalized. This method produces optimal results when the computation of depth values is performed for both human impostors and their accessories, as shown in Figure 5.3 (c). Also, note that the presented accessory impostor method is also compatible with more sophisticated virtual human impostor methods such as the one of [Kavan et al., 2008].

5.3 Accessories Implementation

In this Section, we focus on the architectural aspect of accessories. We present with small pseudocode snippets, how to store, load, and render them.

5.3.1 Loading and Initialization

First of all, each accessory has a type, *e.g.*, “hat” or “back pack”. We empirically differentiate seven different types. In order to avoid the attribution of, for instance, a cowboy hat and a cap on the same head, we never allow a character to wear more than one accessory of each type. To distribute accessories to the whole crowd, we need to extend the following data structures (introduced in Section 3.2):

- **Human template:** each human template is provided with a list of ids corresponding to the accessories it can wear. This list is sorted by type. This way, we know which template can wear which accessory. This process is necessary, since all human templates cannot wear all accessories. For instance, a school bag would suit the template of a child, but for an adult template, it would look much less believable.

- **Human instance:** each human instance possesses one accessory slot per existing type. This allows to later add up to seven accessories (one of each type) to the same virtual human.

We also create two data structures to make the accessory distribution process efficient:

- **Accessory entity:** each accessory possesses a list of human instance ids, representing the virtual humans wearing it. They are sorted by human template.

- **Accessory repository:** an empty repository is created to hold all accessories loaded from the database. They are sorted by type.

We show a schematic view of the accessory repository in Figure 5.5 (left): all accessories are listed and sorted by type. Each accessory possesses a list of human instance ids using it. At initialization, the above data structures are filled, based on the content of our database (introduced in Section 3.3.2). We detail this process in the three following pseudocode snippets:

```
01 // Create accessory entity and fill accessory repository.
02 For each accessory a in database:
03    load a’s data contained in the database,
04    create a’s vertex buffer (for later rendering),
05    insert a into the accessory repository (sorted by type).
```

```
01 // Fill each human template’s accessory list.
02 For each human template h:
03    For each accessory a suitable to h :
04       insert a’s id into h’s list l (sorted by type).
```

```
01 // Fill each human instance’s accessory slots and
02 // fill each accessory’s list of instances wearing it.
03 For each human instance i:
04    get human template h of i,
05    get accessory id list l of h,
06    For each accessory type t in l:
07       choose randomly an accessory a of type t,
08       assign a to the correct accessory slot of i,
09       push i’s id in a’s human instance id list (sorted by human template).
```

The process of filling these data structures is done only once at initialization, because we assume that once specific accessories have been assigned to a virtual human, they never change. However, it would be easy to change the accessories worn at runtime, through a call to the last loop. Note that a single vertex buffer is created for each loaded accessory, independently from the number virtual humans wearing it.

5.3.2 Rendering

Since the lists introduced in the previous section are all sorted according to our needs, the rendering of accessories is much facilitated. We show in the following pseudocode our pipeline:

```
01 For each accessory type t of the repository:
02    For each accessory a of type t:
03       bind vertex buffer of a,
04       send a’s appearance parameters to the GPU,
05       get a’s list l of human instance ids (sorted by human template).
06       For each human template h in l:
07          get the joint j of h to which a is attached,
08          get the original position matrix m1 of j,
09          get the displacement matrix m2 of couple [a,h],
10          For each human instance i of h:
11             get matrix m3 of i’s current position,
12             get matrix m4 of j’s current deformation for i,
13             multiply current modelview matrix by mi (i=1..4),
14             call to vertex buffer rendering.
```

This pseudocode is especially optimized in order to minimize state switches. First of all, at line 03, each accessory has its vertex buffer binded. We can process this way, independently from the human instances, because an accessory never changes its shape or texture.

Then, we process through each accessory's human instance id list (line 05). This list is sorted by human template (line 06), allowing us to retrieve information common to all its instances, *i.e.*, the joint j to which is attached the accessory (line 07), along with its original position matrix $m1$ in the skeleton (line 08), and the original displacement matrix $m2$ between $m1$ and the desired position of the accessory (line 09). An example illustrating $m1$ and $m2$ with a hat attached to the head joint of two human templates is shown in Figure 5.1.

Once the human template data is retrieved, we iterate over each instance wearing the accessory (line 10). A human instance also has specific data that is required: its position for the current frame, in matrix $m3$ (line 11), and the displacement of its joint, relatively to its original posture, depending on the animation played, in matrix $m4$ (line 12). Figure 5.4 illustrates the transformation represented by these two matrices.

Finally, by multiplying the matrices extracted from the human template and instance, we are able to define the exact position and orientation of the accessory (line 13). The rendering of the vertex buffer is then called and the accessory is displayed correctly (line 14).

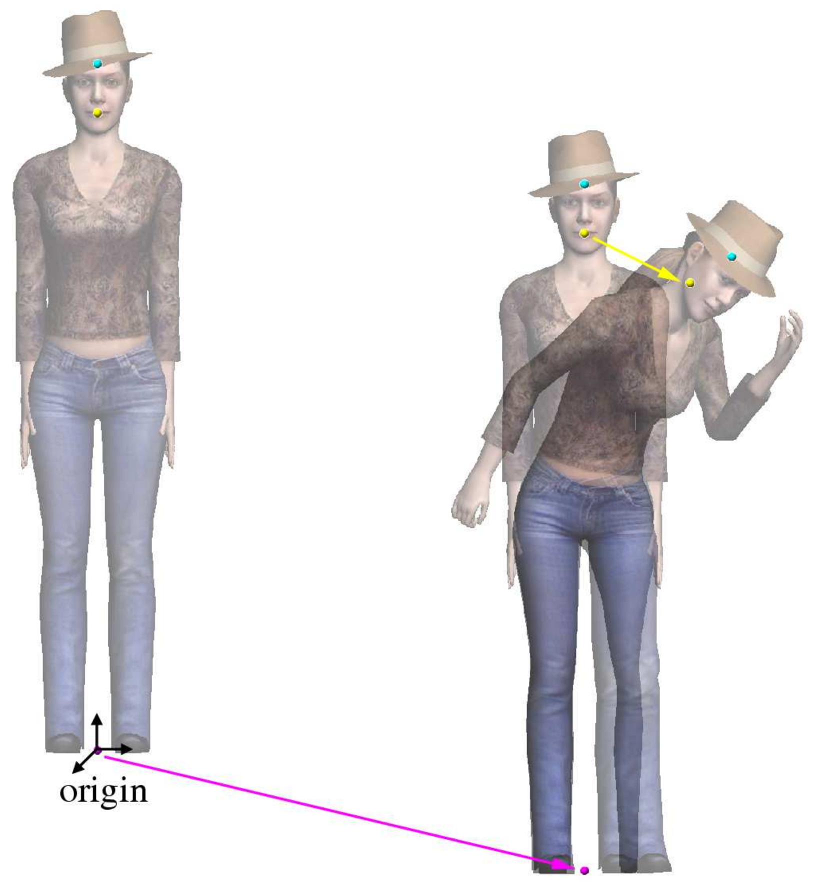


Figure 5.4: *(Left)* a human template in default posture. *(Right)* An instance of the human template playing an animation clip. The displacement of the body, relatively to the world origin $(m3)$ is depicted in red, the displacement of the head joint due to the animation clip $(m4)$ in yellow.

5.3.3 Empty Accessories

We have identified seven different accessory types. And, through the accessory attribution pipeline, we assign seven accessories per virtual human. This number is quite large and the results obtained can be unsatisfying: indeed, if all characters wear a hat, glasses, jewelry, a back pack, *etc.*, they look more like christmas trees than believable people. We need the possibility to have people without accessories too. To allow for this, we could simply randomly choose for each human instance's accessory slot, whether it is used or not. This solution works, but a more efficient one can be considered. Indeed, at the rendering phase of a large crowd, testing each slot of each body to know whether it is used or not implies useless code branching, *i.e.*, precious computation time.

We therefore propose a faster solution to this problem by creating empty accessories. An empty accessory is a fake one, possessing no geometry nor vertex buffer. It only possesses a unique id, similarly to all other accessories. At initialization, before loading the real accessories from the database, the following pseudocode is executed:

```
01 For each accessory type t:
02    create one empty accessory e of type t,
03    put e in the accessory repository (sorted by type),
04    For each human template h:
05       put e's id in h's accessory id list.
```

The second loop over the human templates at line 04 is necessary in order to make all empty accessories compatible with all human templates. Once this pre-process is done, the loading and attribution of accessories is achieved as detailed in Section 5.3.1. This preliminary introduction of empty accessories causes later their possible insertion in some of the accessory slots of the bodies. Note that if, for instance, a body entity gets an empty accessory for hat, reciprocally, the id of this body will be added to the empty accessory's body id list. This is illustrated with an example in Figure 5.5 (right), where human instance 1 (referred to as $Body1$) wears an empty accessory in the glasses category. One may wonder how the rendering of an empty accessory is achieved. If keeping the same pipeline as detailed in Section 5.3.2, we meet troubles when attempting to render an empty accessory. Moreover, some useless matrix computations would be done. Our solution is simple. Since the empty accessories are the first ones to be inserted into the accessory repository (sorted by type), we only need to skip the first element of each type to avoid their computation and rendering. The pseudocode given in Section 5.3.2 only needs a supplementary line, which is:

```
01b skip first element of t.
```

With this solution, we take full advantage of accessories, obtaining varied people, not only through the vast choice of accessories, but also through the possibility of not wearing them. And there is no need for expensive tests within the rendering loop. In Figure 5.6, we show the results obtained with a single human template instantiated several times, using accessories in addition to the appearance variety detailed in Chapter 4.

5.3.4 Color Variety Storage

In Chapter 4, we detailed how to apply color variety to the different body parts of a texture. The same method can be applied to the accessories. A human texture is segmented in eight

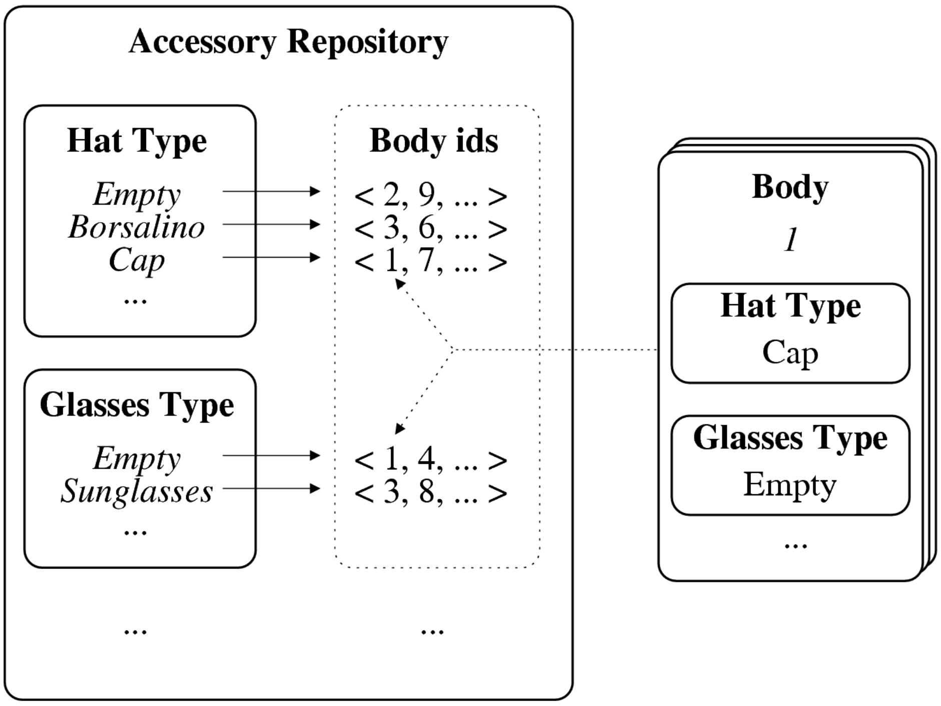


Figure 5.5: *(Left)* a representation of the accessory repository, sorted by type. Each accessory possesses its own list of body (or human instance) ids. Reciprocally, all bodies possess slots filled with their assigned accessories. *(Right)* an illustrated example of the accessory slots for body with id 1.

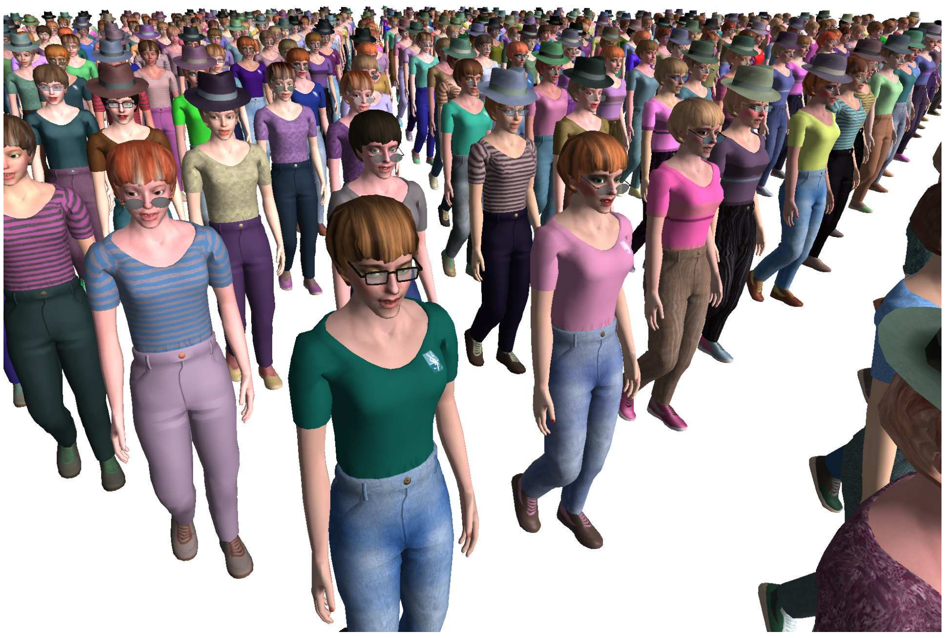


Figure 5.6: Several instances of a single human template, varied through the appearance sets and several segmentation maps, and accessories.

body parts, each having its specific color range. At initialization, for each instantiated virtual human and each body part, a color is randomly chosen in a range to modulate the original color of the texture.

Since accessories are smaller and less complex than virtual humans, we only use four different parts, *i.e.*, one segmentation map per appearance set. Then, similarly to the characters, each instance of each accessory is randomly assigned four colors within the HSB ranges defined for each part. These four random colors have also to be stored. We reemploy the Color Look-Up Table (CLUT) of human instances to save the colors of accessories. In order not to confuse the color variety of the body parts and those of the accessories, we store the latter contiguously from the bottom-right of the CLUT (see Figure 4.11), whereas the human instance colors are stored from the top-left. Each character thus needs eight texels for its own color variety and 7×4 other texels for all its potential accessories. This sums up to 36 texels per character. A 1024×1024 CLUT is therefore able to roughly store more than $29,000$ unique color variety sets.

5.4 Accessory Limitations

The accessories presented above are a nice solution to further add variety to a crowd of human instances. They are simple to use, scalable, and provide visually appealing results. Nevertheless, to keep the accessorizing process as simple as possible, we have made assumptions that limit our technique.

Firstly, the mesh is presumed to be attached to a single joint, limiting the possibilities. It would be possible to skin an accessory with more joints, but adapting it to any template without changing the mesh vertices would prove to be difficult. Secondly, to attach accessories to moving characters, we assume to work with skeletons and skeletal animations, which is not necessarily the case. Nevertheless, this limitation can easily be overcome by attaching an accessory, for instance, to one of the vertices composing the character instead of a joint.

Thirdly, the technique does not provide solutions for simulating movements independent from the attach joint, *e.g.*, a hat too big sliding on a child's head. However, this could be implemented as a supplementary layer on top of the current positioning algorithm. Finally, some accessories cannot be used as presented here, because their presumed weight should alter the animation of the characters. For instance, a handbag can easily be placed in a virtual human's hand, but if the performed animation sequence is not altered, the bag seems weightless, and the resulting effect is not realistic. We have come up with two solutions to solve this issue, which are presented in the next chapter. Moreover, the range of accessories unconcerned by this limitation is sufficiently large to already obtain unique instances in crowds.

5.5 Shape and Height

A second possibility to add shape variety to instances of a same human template is to modify their morphology, like their height and shape. A dedicated tool has been created in order to help a designer for such a task: for each template, the designer can specify skeleton and skinning alterations. Skeleton modifications allow for global height variations and more localized effects like broad / narrow shoulders, while skinning deformations increase the shape uniqueness by scaling each vertex independently to obtain thin, fat, muscular and pregnant templates. Combined shape and height effects are illustrated in Figure 5.7.

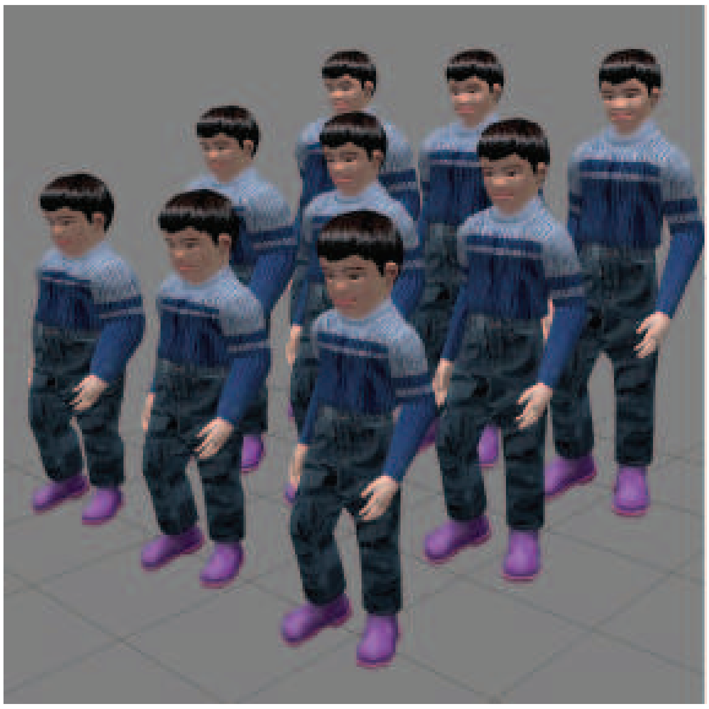


Figure 5.7: The space of interactive height and shape variety at creation. The human template grows up in the right axis, while it gets more fat on the left axis.

5.5.1 Shape

Modifying the shape of a human mesh is achieved in three steps.

Step 1. Using a commercial 3D package like 3DSMax [Autodesk, 2009a], it is possible for a designer to paint a *FatMap* for a given template, as seen in Figure 5.8. The FatMap is an extra gray-scale UV texture that is used to emphasize body areas that store fat. Darker areas represent regions where the skin will be most deformed, *e.g.*, on the belly, and lighter areas are much less deformed, like the head. When the creation of the FatMap is complete, the grayscale values at each texel are used to automatically infer one value for each vertex of the template's mesh. Each of these values, called a *fatWeight*, is attached to the vertex as an additional attribute.

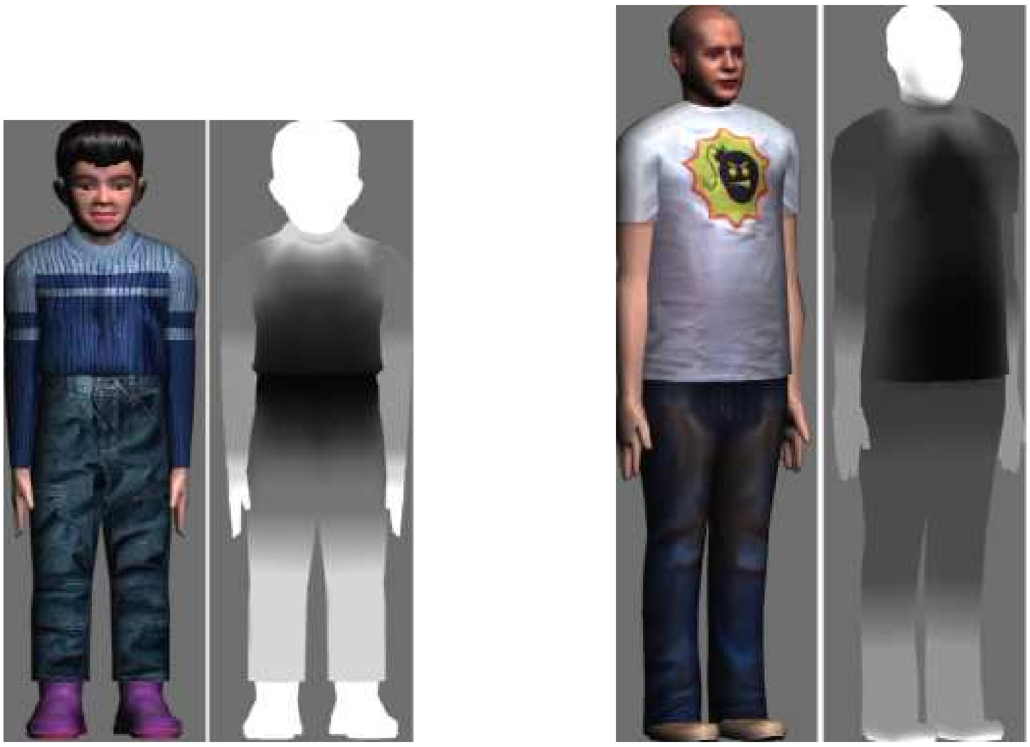


Figure 5.8: Two FatMaps designed in 3DSMax [Autodesk, 2009a]. Dark areas represent regions more influenced by fat or muscle modification, while lighter parts are less modified.

Step 2. The next step is to compute in which direction the vertices are moved when scaled. We have found that using the normal of a vertex as its scaling direction provided bad results, especially at the shoulders and armpits. Instead, we compute the scaling direction of each vertex as the weighted normal of the bones influencing it: for each joint j influencing a vertex v, we first compute the vector $\vec{j}$, *i.e.*, the vector connecting j to its child j_2 (as $j_2 - j$). Second, we compute the vector connecting j_2 to v as $\vec{j_2 v} = v - j_2$. Finally, we can obtain the normal $\vec{n_j}$ to $\vec{j}$ that passes through v as:

$$\vec{n_j} = \vec{j_2 v} - \left(\hat{j} \cdot \frac{\vec{j} \cdot \vec{j_2 v}}{\|\vec{j}\|} \right). \tag{5.5}$$

Where $\hat{j}$ is the unit vector in direction $\vec{j}$. The final vector n_v, normal to v, is computed as the sum of normals computed with Equation 5.5 for all joints influencing v, weighted by their level of influence (which is provided from the skinning phase of the vertex):

$$\vec{n_v} = \sum_{j=0}^{4} \hat{n_j} \cdot weight_j \tag{5.6}$$

where $\vec{n_j}$ is the unit vector in direction n_j, computed with Equation 5.5, and $weight_j$ is the level of influence of joint j on vertex v. Note that the above equations cannot be computed if j is a childless joint. In such a case, we use an approximation to evaluate n_j as $v - j_w$, where j_w is the position of joint j in the world coordinates.

Step 3. Once the direction of the body scaling is computed for each vertex, the actual scaling can take place. The extent to which we scale the body is defined by a *fatScale*, randomly chosen within a pre-defined range. Each vertex v is thus deformed as:

$$v' = v + (\vec{n_v} \cdot (1 - fatWeight_v) \cdot fatScale) \tag{5.7}$$

where $fatWeight_v$ is the weight associated to v at the creation of the FatMap (see Step 1).

A design tool has been created in order to help a designer in the vertex editing phase. It allows the designer to test the FatMap created in another software, and to decide of minimal and maximal values to which vertices are authorized to be scaled, *i.e.*, the values *fatScale* can take.

5.5.2 Height

Our second contribution is to modify the height of a human template, by scaling its skeleton. To help the designer in this task, we provide additional functionalities to the design tool presented above: for a given human template skeleton, the global space of height scaling can be defined. Fine-grained local tuning for each joint can also be specified, *i.e.*, minimal and maximal scale parameters on the x, y, and z world axes. These data allow several different skeletons to be generated from a single template, which we call the ***meta-skeleton***. For each new skeleton, a global scale factor is randomly chosen within the given range. Then, the associated new scale for each of its bones is deduced. Short / tall skeletons mixed with broad / narrow shoulders are thus created.

The skin of the various skeletons also needs adaptation. Each vertex v of the original template is displaced by each joint j that influences it:

$$\begin{aligned} v' = v + (&worldMatrixSkeleton_j \\ &\cdot (inverseWorldMatrixMetaSkeleton_j \cdot v \cdot scale_j)) \end{aligned}$$

where $inverseWorldMatrixMetaSkeleton_j$ is the matrix that allows to express vertex v in the original joint reference, and $worldMatrixSkeleton_j$ is the transformation matrix of joint j in the newly generated skeleton.

Both methods, skeleton modification and displaced skin vertices, are compatible when applied consecutively. Some results of the technique applied to a template are illustrated in Figure 5.9. The main limitation to this approach is that it is not compliant with other levels of detail (LOD) at low cost. In both cases, our current solution is to generate a series of skeletons and meshes in a pre-process, and compute rigid meshes / impostors for each of them. Unfortunately, this solution is quite limiting in terms of memory. Finding solutions to efficiently scale this work is our main concern for future work.

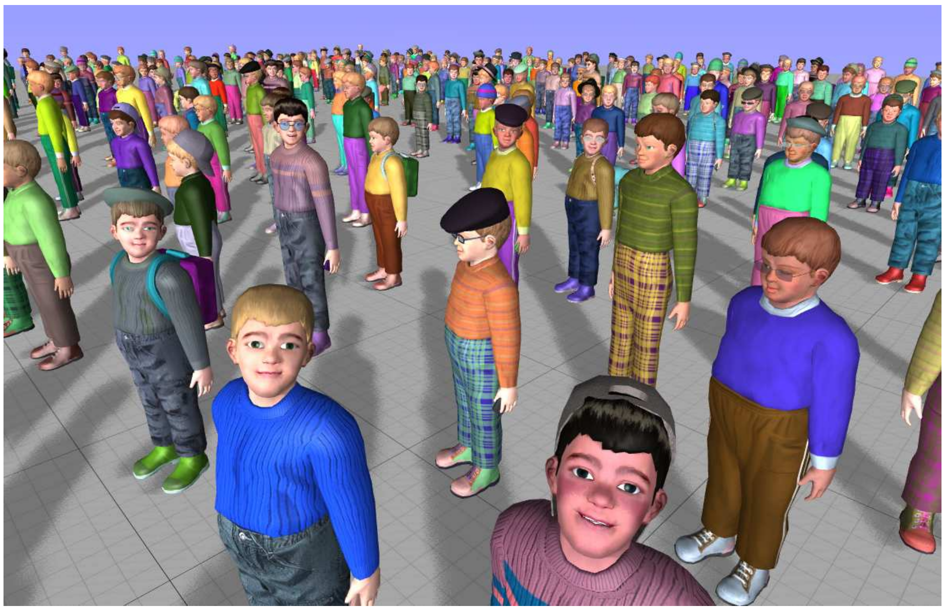


Figure 5.9: Results of shape and height modifications shown on multiple instances of a single template.

CHAPTER 6

Animation Variety

As explained in previous chapters, it is possible to vary the appearance and shape of individuals, even when issued from the same human template. However, we introduced in Chapter 3 the necessity to also provide a large variety of animation clips to the simulation. The most important factor for virtual humans to look different is to modify their visual appearance, *i.e.*, their body part colors, and their shape. A second important factor, although less paramount is their animation. If they all perform the same animation, the results are not realistic enough [McDonnell et al., 2008]. In this chapter we describe three techniques to vary the animation of navigating characters, *i.e.*, working with locomotion animations. First, in Section 6.1, we introduce variety in the animation by generating a large amount of *locomotion* cycles (walking and running), and *idle* cycles (like standing, talking, sitting, *etc.*). In Section 6.2, we detail the *motion kit*, a data structure, previously introduced in Chapter 3, that efficiently handles animations at all levels of detail (LOD). Then, in Section 6.3, we present a second technique of animation variety, *i.e.*, how pre-computed animation cycles can be augmented with upper-body variations, like having a hand on the hip, or in a pocket. Finally, in Section 6.4, we introduce the third technique to achieve variety: procedural modifications applied at runtime on locomotion animations to allow crowds to wear complex accessories (introduced in Section 5.1.2).

6.1 Animation Types

In order to obtain variety in animation, there is a great need for a large set of raw animation cycles that can then be further varied. Varied animations have to be created for each human template, and stored on a centralized animation clip database to be used directly by the human instances.

For a typical crowd scenario, we create two kinds of clips: *idle* and *locomotion* clips, that we morphologically adapt for each template. Idle clips are usually hand-designed. We take care to make these animations cyclic, and categorize them in the database, according to their type: sitting or standing, talking or listening, *etc*. For locomotion clips, walk and run cycles are generated from a locomotion engine based on motion capture data. We compute such locomotion clips for a set of speeds. Thus, during real-time animation, it is possible to directly obtain an adequate animation for a virtual human, given its current locomotion velocity, and its morphological parameters.

Figure 3.1 presents a convenient schema to visualize and situate the various animation components (in the Variety box), within the overall *YaQ* architecture.

6.1.1 Idle Animation Clips

Since idle motion clips are created by a designer, they are per se very different, and no additional variety technique is required to make them unique. This is not the case for locomotion cycles, which are further discussed in the remaining of this chapter.

Idle clips are used in specific cases. At the initialization of *YaQ*, it is possible for the user to indicate a series of navigation graph vertices, where he would like to see idle pedestrians. The reader is invited to refer to Section 10.1 of the Appendix for an introduction on navigation graphs. The user thus provides a list of graph vertices, the number of pedestrians he wishes to put there, and the type of idle animations he wants the pedestrians to play. Then, at runtime, and thanks to the meta-information associated to each idle clip in the database, pedestrians are randomly picked and inserted in the chosen graph vertex, and they start playing the type of idle animations that has been assigned to them. Care is taken to make idle clips cyclic, and that their starting/ending frame is the same for all clips of the same type (having the same meta-information). Thus, when an idle pedestrian has finished playing an animation, it can randomly pick another animation of the same type without any transition problem. We do not further detail this type of animations here, as they are sufficiently varied to not require any further treatment.

6.1.2 Locomotion Animation Clips

We recall here the locomotion engine of [Glardon et al., 2004a,b] that we have used to generate our original set of walk and run cycles.

Glardon *et al.* have introduced a PCA-based locomotion engine capable of animating on the fly human-like characters of any size and proportion by generating complete locomotion cycles. They have captured walk and run motions from several people, from which they have created a normalized model. There are mainly three high-level parameters which allow to modulate these motions:

- Personification weights: five people, different in height and gait have been captured while walking and running. This variable allows the user to choose how he wishes to parametrize these different styles.

- Speed: the five subjects have been captured at different speeds. This parameter allows to choose at which velocity the walk/run cycle should be generated.
- Locomotion weights: this parameter defines whether the cycle is a walk or a run animation.

Thus, the engine is able to generate a whole range of varied locomotion cycles for a given character. To efficiently animate the locomotion of each individual, we generate in a pre-process a certain number of locomotion cycles for each human template. We have used this engine to generate over 100 different locomotion cycles per human template: for each one of them, we sample walk cycles at speeds varying from $0.5\ m/s$ up to $2\ m/s$ and, similarly for the run cycles, between $1.5\ m/s$ and $3\ m/s$. Each human template is also assigned a particular personification weight so that it has its own gait. With such a high number of animations, we are already able to perceive a sense of variety in the way the crowd is moving. Virtual humans walking together with different locomotion styles and speeds add to the realism of the simulation.

6.2 Motion Kits

We have developed three levels of representations for the virtual humans: deformable meshes, rigid meshes, and impostors. When playing an animation sequence, a virtual human is treated differently depending on its current distance and eccentricity to the camera, *i.e.*, the current LOD it uses. For clarity purpose, we have given each animation clip a different name depending on which level of detail it applies to: an animation clip intended for a deformable mesh is a *skeletal animation*, one for a rigid mesh is a *rigid animation*, and finally, an animation clip for an impostor is an *impostor animation*.

We have already shown that the main advantage of using less detailed representations is the speed of rendering. However, for the memory, the cost of storing an animation sequence for a deformable mesh *vs.* a rigid mesh *vs.* an impostor is increasingly expensive (see Figure 7.4). From this, it is obvious that the number of animation sequences stored must be limited for the less detailed representations. It is also true that we want to keep as many skeletal animation clips as possible for the deformable meshes, firstly, because their storage requirement is cheap, and secondly, for variety purposes. Indeed, deformable meshes are at the forefront, close to the camera, and several virtual humans playing the same animation clip are immediately noticed.

The issue arising is then switching from a level of representation to another. For instance, what should happen if a deformable mesh performing a walk cycle reaches the limit at which it switches to the rigid mesh representation? If a rigid animation with the same walk cycle (same speed) has been pre-computed, switching is done smoothly. However, if the only rigid animation available is a fast run cycle, the virtual human will “pop” from a representation to the other, which is a disturbing artifact that may attract the eye of the observer. We therefore need each skeletal animation to be linked to a ressembling rigid animation, and similarly to an impostor animation. For this reason, we have developed the *motion kit* data structure. We first describe the motion kit data structure in Section 6.2.1 and then its implementation in Section 6.2.2.

6.2.1 Data Structure

A motion kit holds several items:

- A name, identifying what sort of animation it represents, *e.g.*, *walk_1.5*,
- Its type, determined by four identifiers: *action*, *subaction*, *left arm action*, and *right arm action*,
- A link to a skeletal animation,
- A link to a rigid animation,
- A link to an impostor animation.

The only knowledge stored in each virtual human instance is the current motion kit it uses. Then, at the Animator stage of the runtime pipeline (see the Real-time Component in Figure 3.1), depending on the distance of the virtual human to the camera, the correct animation clip is used. Note that there is always a 1:1 relation between a motion kit and a skeletal animation, *i.e.*, a motion kit is useless if there is no corresponding skeletal animation. As for the rigid and impostor animations, their number is much smaller than for skeletal animations, and thus, several motion kits may point to the same rigid or impostor animation. For instance, imagine a virtual human using a motion kit representing a walk cycle at $1.7\ m/s$. The motion kit has the exact skeletal animation needed for a deformable mesh (same speed). If the virtual human is a rigid mesh, the motion kit may point to a rigid animation at $1.5\ m/s$, which is the closest one available. And finally, the motion kit also points to the impostor animation with the closest speed. The presented data structure is very useful to easily switch from a representation to another.

In Figure 6.1, we show a schema representing a motion kit and its links to different animation clips. All the motion kits and the animations are stored in a database, along with the links joining them (see Section 3.3.2). One may wonder what the four identifiers are for. They are used as categories to sort the motion kits. With such a classification, it is easy to randomly choose a motion kit for a virtual human, given certain constraints.

Firstly, the *action type* describes the general kind of movements represented by the motion kit. It is defined as either:

- *idle/stand* for all animations where the virtual human is standing on its feet,
- *idle/sit* for all animations where the virtual human is sitting,
- *locomotion/walk* for all walk cycles, or
- *locomotion/run* for all run cycles.

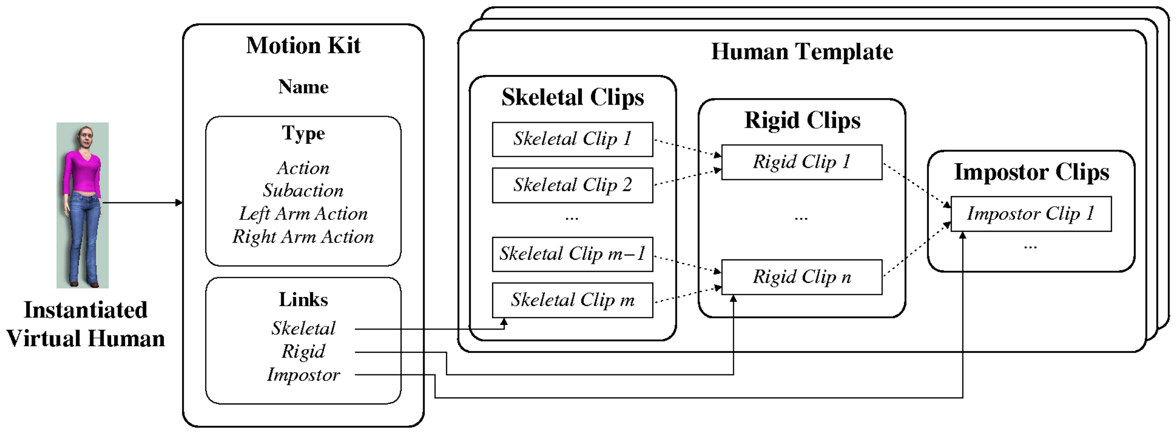


Figure 6.1: Example of motion kit structure. On the left, a virtual human instantiated from a human template points to the motion kit it currently uses. In the center, a motion kit with its links identifying the corresponding animations to use for all human templates and LOD.

The second identifier is the *subaction type*, which more restrains the kind of activity of the motion kit. Its list is non-exhaustive, but it contains descriptors such as: *talk*, *dance*, *listen*, *etc.* We have also added a special subaction called *none*, which is used when a motion kit does not fit in any of the other subaction types. Let us note that some action / subaction couples are likely to contain no motion kit at all. For instance, a motion kit categorized as a *sit* action and a *dance* subaction is not likely to exist.

The third and fourth identifiers: *left* and *right arm actions* are used to add some specific animation to the arms of the virtual humans. For instance, a virtual human can walk with the left hand in its pocket and the right hand holding a cellphone. We further detail how such animation clips are generated in Section 6.3. For now, it is sufficient to know that these two identifiers are used to further categorize a motion kit. There are three options to set these identifiers: *none*, which means that no special arm activity is achieved; *pocket*, indicating that the hand is in its pocket; and *cellphone*, for having the hand close to the ear, as if making a phone call. This list can be extended to other possible arm actions. For instance, holding an umbrella, pull a caster suitcase, or scratch one's head.

When we create a varied crowd with *YaQ*, it is simple for each virtual human to randomly ask for one of all the available motion kits. If the need is more specific, *e.g.*, a virtual human sitting on a bench, it is easy to choose only the adequate motion kits, thanks to the identifiers. To illustrate this, we show an example in Figure 6.2, where a virtual human is playing skeletal animation, linked to a motion kit with the following identifiers: *[walk] [none] [cellphone] [pocket]*.

6.2.2 Implementation

In *YaQ*, at initialization, the motion kits of the chosen human templates are uploaded from the database, and stored in a four-dimensional table:

```
Table[ action id ][ subaction id ][ left arm action id ][ right arm action id ].
```

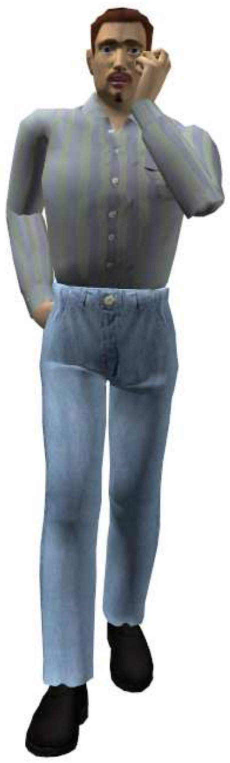


Figure 6.2: A virtual human using a motion kit with identifiers: *[walk] [none] [cellphone] [pocket]*.

For each combination of the four identifiers, a list of motion kits corresponding to the given criteria is stored. As previously mentioned, not all combinations are possible, and thus, some lists are empty.

In our architecture, an animation (whatever its LOD) is dependent on the human template playing it : for a deformable mesh, a skeletal animation sequence specifies how its skeleton is moved, which causes the vertices of the mesh to get deformed on the GPU. Since each human template has its own skeleton, it is impossible to share such an animation with other human templates. Indeed, it is easy to imagine the difference there is between a child and an adult skeleton. For a rigid animation, the already deformed vertices and normals are sent to the GPU. Thus, such an animation is specific to a mesh, and can only be performed by a virtual human having this particular set of vertices, *i.e.*, issued from the same human template. Finally, an impostor animation clip is stored as a sequence of pictures of the virtual human. It is possible to modify the texture and color used for the instances of the same human template, but it seems obvious that such pictures cannot be shared by different human templates. This specificity is reflected in our implementation, where three lists of skeletal, rigid, and impostor animations are stored for each human template.

It follows that each motion kit should also be human template-dependent, since it has a physical link to the corresponding animation triplet. However, this way of managing the data is far from optimal, because usually an animation (whatever its LOD) is always available for all the existing human templates. It means that, for instance, if a template possesses an animation imitating a monkey, all other human templates are likely to have it too in their animation repertoire. Thus, making the information contained in a motion kit human template-dependent would be redundant. We introduce two simple rules that allow us to keep a motion kit independent from a human template:

(1) For any motion kit, all human templates have the corresponding animations.

(2) For all animations of all human templates, there is a corresponding motion kit.

Thanks to these assertions, we can keep a motion kit independent from the human templates. We now explain how to still keep the knowledge of which animation triplet is linked with which motion kit. First, note that each human template contains amongst other things:

- A list of skeletal animations,
- A list of rigid animations,
- A list of impostor animations.

Following the two rules mentioned above, all human templates contain the same number of skeletal animations, the same number of rigid animations, and the same number of impostor animations. If we manage to sort these animation lists similarly for all human templates, we can link the motion kits with them by using their index in the lists. We show a simple example in Figure 6.1, where a structure representing a human template is depicted. On the left-side of the image, a motion kit is represented, with all its parameters. Particularly, it possesses three links that indicate where the corresponding animations can be found for all human templates. These links are represented with arrows in the figure, but in reality, they are simply integers that can be used to index each of the three animation lists for all human templates.

With this technique, we are able to treat all motion kits independently from the human templates using them. The only constraint is to respect rules (1) and (2).

6.3 Upper Body Movements

Further variations in locomotion clips are introduced in order to increase individuality in motion. Indeed, in reality, individuals composing a crowd are rarely walking the same way, arms resting alongside the body (the exception would be military march activities). Most of the time, hands are used to hold objects (cellphone), are hidden in clothes (pocket), or simply rest on the hip. Upper body variations are thus introduced in the cycles which keep the forward-backward movement of the pelvis when walking and running. Example of such variations are visible in Figure 6.3.

Further variations in motion clips are introduced in order to increase individuality in motion. Indeed, in reality, individuals composing a crowd are rarely walking the same way, arms resting alongside the body (the exception would be military march activities). Most

of the time, hands are used to hold objects (cell phone, bag, flowers), are hidden in clothes (pocket), or simply rest on the hip. Upper body variations are thus introduced in the cycles which keep the forward-backward movement of the pelvis when walking and running. A varied locomotion cycle is created in a pre-process of three steps: first, we generate the cycle with default upper body animation. Then, the designer manually defines a set of constraints, which enforce a specific arm posture, *e.g.*, forcing some joints of the hand to reach a goal position attached to the pelvis to simulate a hand in a pocket. Finally, using an IK solver, we iterate over each frame of the original cycle to obtain the desired arm position.

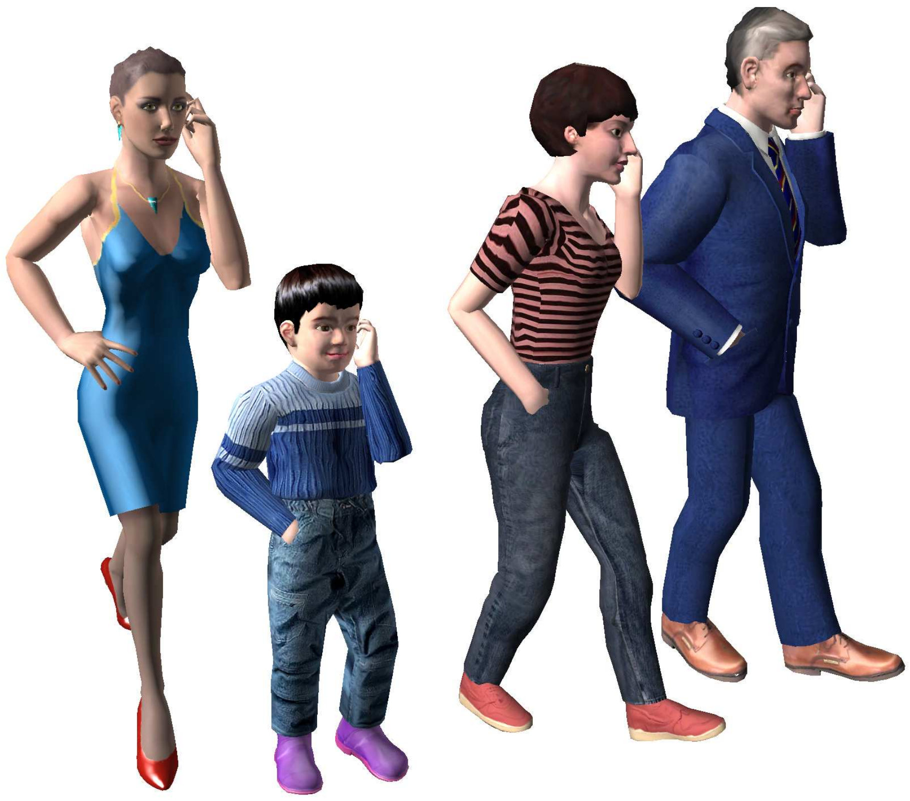


Figure 6.3: Examples of upper body movements (hands in the pocket, phone call, hand on hip, ...) added to a generic locomotion cycle.

The task of augmenting a locomotion cycle with upper body movements, such as those illustrated in Figure 6.3, proves to be more difficult than what it looks at first sight. Indeed, simply blocking the arm of a virtual human in a certain position is not sufficient; since the character is walking, such an approach would result in the hand colliding with other body parts. For upper-body motions to be correctly introduced, they need to be adapted at each animation keyframe in order to follow the movements of the primary locomotion cycle. Such a task is too costly to be achieved in real time, and thus needs to be achieved in a pre-process, *i.e.*, based on the initial locomotion cycle, a new cycle must be generated.

A varied locomotion cycle is created in two passes. The first pass generates the primary cycle with default upper-body animation. The second pass, consisting in adding the secondary upper-body motion, is achieved using a prioritized Inverse Kinematics (IK) solver [Baerlocher and Boulic, 2004]. This tool allows to enforce several constraints at the same time, with levels of priority if necessary. To solve the IK problem, the following inputs are needed:

- The original locomotion cycle;
- A hand-designed "first guess" posture of the hand and arm, using dedicated software, such as MotionBuilder [Autodesk, 2009b];
- The set of constraints to apply to the hand and/or arm. Each constraint is described with two points on the body that we require to be placed as close to each other as possible. For instance, in Figure 6.4, the three colored cubes on the hand have to be positioned as close as possible to the three corresponding colored cubes attached to the pelvis.

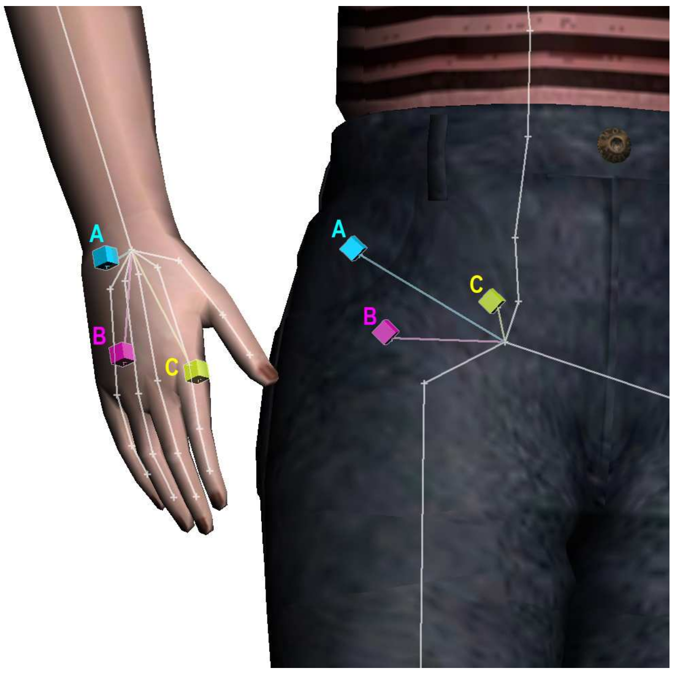


Figure 6.4: Set of controlled effectors attached to the hand and corresponding goal positions attached to the pelvis.

Once the inputs are provided, the IK solver is run for each frame of the animation, starting with the first guess posture of the arm. When all frames have been computed, the final orientation of the modified joints are used to overwrite the original orientations. We illustrate in Figure 6.5 a locomotion cycle that has been augmented with the right hand in the pocket.

In order to make these variations scalable to all levels of detail, we need to pre-compute some rigid animations, exactly as we do for primary locomotion cycles. For the impostors however, we have observed that there is no need to pre-compute such variations, for they are too subtle to be noticed at far distances. Thus, when a virtual human transits from a rigid animation to an impostor animation, its upper body animation (if it uses one) is switched to a standard locomotion, with arms alongside the body.

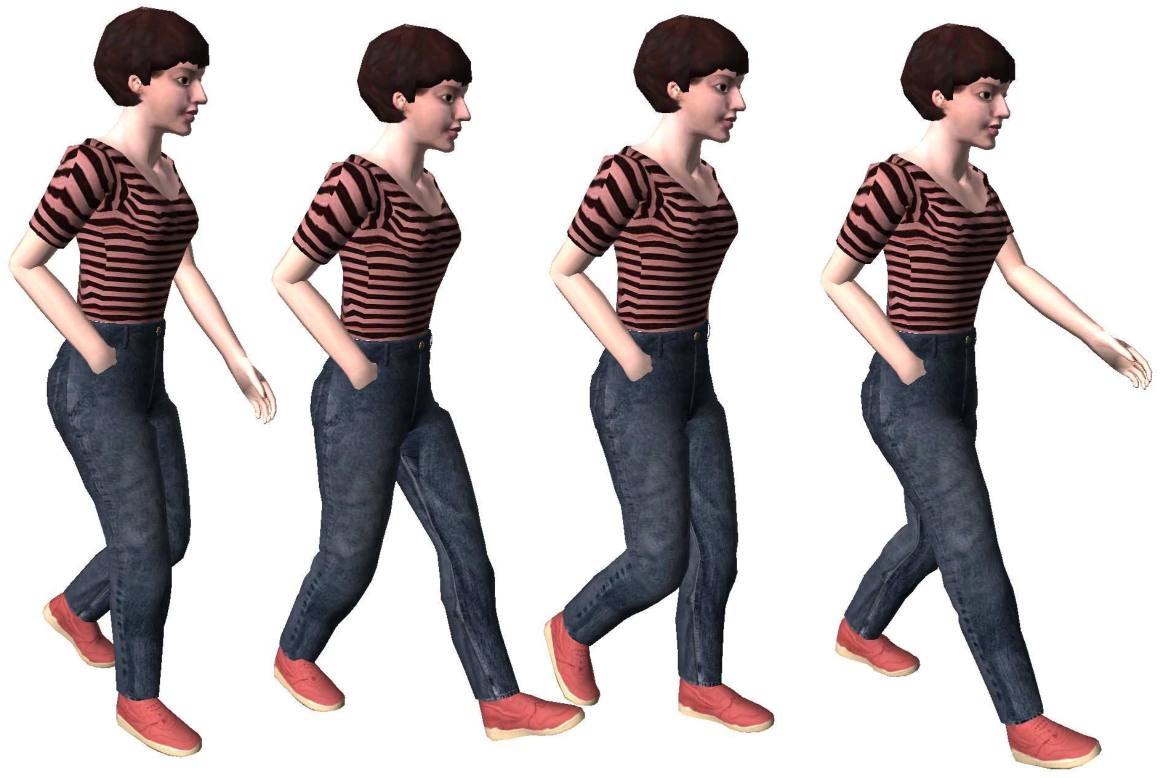


Figure 6.5: Example of posture from an accessorized locomotion cycle.

6.4 Complex Accessories

The main drawback of upper-body variations as presented in the previous section is that they need to be pre-computed and stored in the animation database as extra-animations. In some simpler cases, the use of an IK solver is not necessary, and applying some online modifications is sufficient to obtain realistic results. This is the case for most complex accessories (bags, flowers, boxes, suitcases, *etc.*), except for the cellphone, where the hand needs to stay close to the ear. In this case the technique of Section 6.3 is used.

Similarly to simple accessories, complex accessories are attached to a single skeleton joint. Also, the accessory attribution and rendering process remains the same for both types. The main changes happen at the Animator stage of the runtime pipeline (see Figure 3.1). The use of complex accessories is processed in four steps.

Step 1. At initialization, and for each complex accessory, the designer can specify which joints will be constrained and in which way. We have implemented two possibilities to constrain a joint online: it can be frozen in a chosen orientation, or its movement can be limited within a given range of angles. For instance, carrying a bunch of flowers requires the shoulder movement is to be limited to a chosen angle range, while the elbow is completely frozen at an angle of about 90 degrees.

Step 2. At runtime, during the animation phase, joints are first updated as usual, based on the current animation keyframe. Once the skeleton posture is fully updated, the joints that need special modifications are treated in the next two steps.

Step 3. The frozen joints are the most easy to update: whichever the state of their current animation matrix (computed in Step 2), we completely overwrite it with the chosen orientation. Note that care is taken to keep the same joint's translation in the matrix, because it determines the length separating the joint to its child, and should never be modified.

Step 4. For joints with constrained orientation ranges, the process is more difficult. Indeed, an orientation matrix often defines a rotation on several axes, and finding out the overall angle of its movement is not intuitive. To solve this problem, we first transform the matrix of Step 2 into an exponential map. An exponential map is a structure composed of three floats, like a 3D vector, which are able to represent any 3D rotation. The advantage of this representation is that its norm corresponds to the exact angle of the described rotation. It is thus much easier to clamp a rotation expressed with an exponential map than with a matrix: first, we take the norm n of the exponential map e and clamp it within the minimal and maximal authorized angles (provided at Step 1) as: $n = clamp(n, min_{angle}, max_{angle})$. Then, the clamped exponential map e' is recomputed as $e' = \hat{e} \cdot n$, where $\hat{e}$ is the normalized vector represented by e. Finally, the rotation matrix is derived from e' and used to overwrite the one that was computed in Step 2.

We illustrate the results obtained with this approach in Figures 6.6 and 6.7. The main drawback of this approach is the difficulty we have to adapt it for rigid meshes and impostors. Indeed, no online animation can be performed on these representations. A first solution is to pre-compute the constrained animations, but the underlying memory requirements would soon be too demanding. Another possibility is to use an alternative representation to rigid meshes and impostors. For instance, using a polypostor, as described in [Kavan et al., 2008], would solve our problem, since its 2D polygon can be updated at runtime.

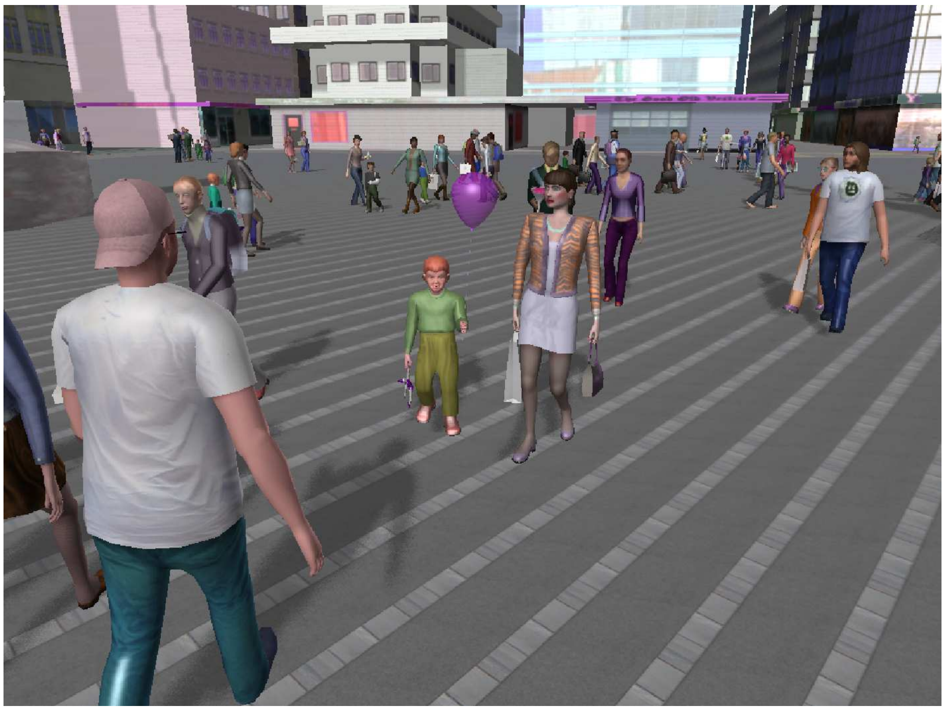


Figure 6.6: Results obtained in an urban environment when modifying the upper body at runtime to make virtual humans carry bags, puppets, balloons, flowers, etc.

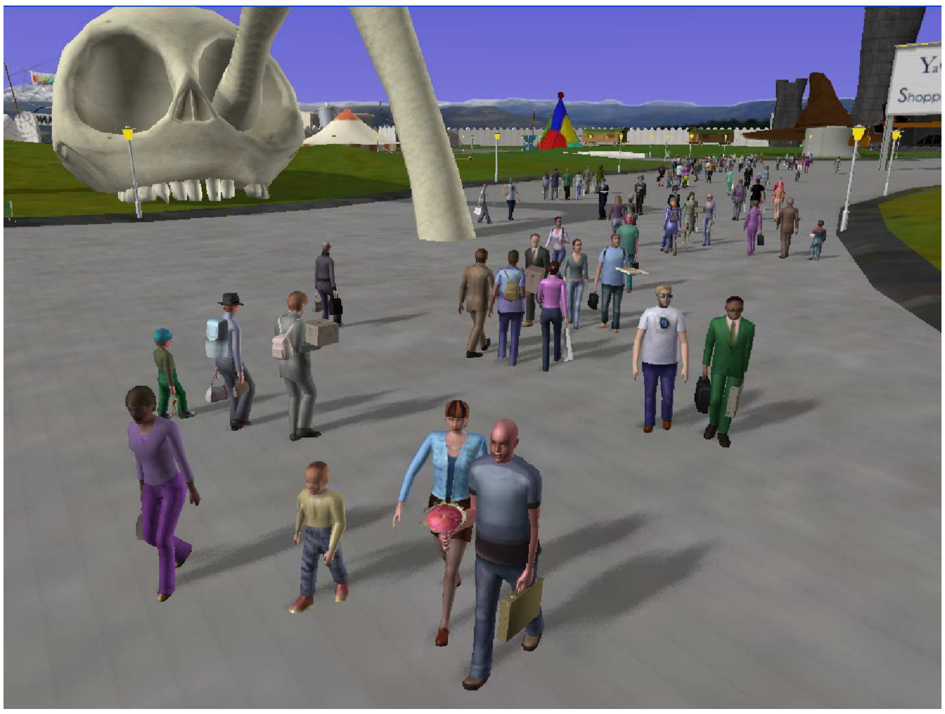


Figure 6.7: More complex accessories carried by virtual humans in a theme park environment.

CHAPTER 7

Real-Time Pipeline

In the Real-time component of *YaQ* architecture (schematized in Figure 7.1), decomposing the process that occurs at each frame into stages allows to efficiently handle the data needed to simulate thousands of characters in real time. A second important point to satisfy real-time constraints is to be able to group similar data and process them together in order to limit costly CPU and GPU state switches. Each stage becomes responsible for a specific task and thus can be developed, tested and optimized separately. It is important to note that in the following description of the Real-Time pipeline, each stage has direct access to the instances composing the crowd.

We distinguish four different stages in *YaQ* pipeline: the Scaler, detailed in Section 7.1, the Simulator, presented in Section 7.2, the Animator (Section 7.3), and finally, the Renderer, introduced in Section 7.4.

7.1 Scaler

The Scaler is the first stage of the pipeline. The work done in this stage consists in finding which simulation and rendering level-of-detail is used for which area of the scene for the current simulation frame.

7.1.1 Score Allocation

The Scaler receives two inputs: a navigation graph filled with virtual human ids and a camera view frustum. From these inputs, the Scaler's role is to provide each navigation graph vertex

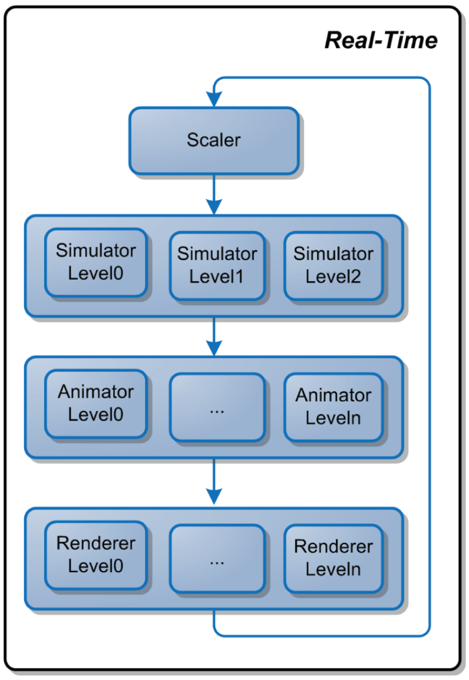


Figure 7.1: The real-time component of *YaQ* architecture is composed of four steps: the Scaler, the Simulator, the Animator, and the Renderer.

with two scores. Firstly, a level of detail (LOD), determined by finding the distance from the vertex to the camera and its eccentricity from the middle of the screen. This LOD score is then used to choose the appropriate virtual human representation in the vertex. Secondly, the Scaler associates with each vertex a score of interest, resulting in an environment divided into regions of different interest (ROI). For each region, we choose a different motion planning algorithm. Regions of high interest use accurate, but more costly techniques, while regions of lower interest may exploit simpler methods.

Using the navigation graph as a hierarchical structure to provide virtual humans with scores is an efficient technique that allows to avoid testing individually each character. The processing of data is achieved as follows: firstly, each vertex of the graph is tested against the camera view frustum, *i.e.*, frustum culled. Empty vertices are not even scored, nor further held in the process for the current frame; indeed, there is no interest to keep them in the subsequent stages of the pipeline. On the other hand, vertices filled with at least one character and outside the camera view are kept, but they are not assigned any LOD score, since they are outside the view frustum, and thus, their virtual humans are not displayed. As for their ROI score, they get the lowest one: a minimal simulation sporadically moves the related virtual humans along their path, and no dynamic collision avoidance need be achieved. This minimal simulation is necessary, even though the characters are invisible, because without care, when they quit the camera field, they immediately stop moving, and thus, get packed on the borders of the view frustum, causing a disturbing effect for the user. Finally, the vertices that are filled and visible are assigned a higher ROI score, and then are further investigated to sort their embedded virtual humans by human template, LOD, and appearance set.

7.1.2 Human Instance Lists

At the end of this first stage, three important human instance lists are obtained, sorted according to different criteria. Updating these lists at each frame takes some time. However, it is very useful to group data in order to process it through the next stages of the pipeline: simple approaches that process virtual humans one after another, in no specific order, provoke costly state switches for both the CPU and GPU. For an efficient use of the available computing power, and to approach hardware peak performance, data flowing through the same path need to be grouped.

To correctly handle virtual human instances, we start by associating a unique identifier to each one of them. Since human instances need to be accessed in many different contexts, several lists of these identifiers are set up, sorted according to different criteria, and kept up-to-date. We mainly distinguish three lists of human instances (or identifiers representing them): the rendering list, the navigation list, and the animation list. All lists are kept up-to-date in the first step of the runtime pipeline, which is detailed in Chapter 7.

Rendering list. This list is sorted to optimize the rendering of all instances. It is sorted according to four criteria:

- By human template. This first criteria seems logical: indeed, instances of a same human template share a lot of common data, *e.g.*, skeleton, mesh, appearance sets.

- By steering type. We distinguish two types of steering which require a different positioning (see Chapter 6: the “locomotion” mode, when pedestrians walk or run, and the “idle” mode, when they perform more quiet actions such as talking, sitting, *etc.*).

- By level of detail. The rendering system is very different to render each type of representation.

- By appearance set. As previously demonstrated, some instances share the same appearance set (but not the same body part colors) and thus, the same texture (more details in Chapter 4.

Note that impostors are very different from deformable or rigid meshes. For this reason, we use a different list to render them, sorted by human template, by appearance set (to avoid constantly switching the texture), by animation, and finally, by keyframe. This animation-based ordering allows us to limit the number of atlas switches: an atlas is read once for all the impostors that use it, before switching to another atlas.

Navigation list. Our navigation list is sorted out to optimize the simulation updates of virtual humans, depending on the region where they are situated, *i.e.*, regions of high, medium, or no interest. Indeed, as introduced in Appendix 10.1, virtual humans are steered with motion planning techniques of various accuracy, depending on their position. We thus use a list sorted with four criteria. First, the level of interest (high, medium, or low), to know which algorithm to use. Second, by graph vertex: each vertex of the navigation graph containing humans is introduced in this list. Third, by steering type (we only navigate the humans in the locomotion mode). A graph vertex can belong to several paths, as later detailed. We thus finally sort the list of human instance according to the path they are following.

Animation list. The main animation of human instances is usually achieved with the same lists used for rendering. An ideal list for this stage would be sorted by human template, by steering type (walking humans are not animated the same way as standing humans, see Chapter 6), and by level of detail. Since the rendering list is exactly the same, except for the appearance set sorting criterion, we do not require to create a new list for animation. The additional work we achieve for facial and procedural animations however, is quite different, because it only applies to deformable meshes. Thus, such additional animations use a much simpler list, indexing only the deformable meshes in no particular order.

In a recent work [Pettré et al., 2006], we have experimented different steering methods. An interesting observation we have made is that with a varying number of characters in a very large scale (tens of thousands), the performance of the different steering methods remained about the same. Memory latency to jump from an instance to the other was the bottleneck when dealing with big crowds.

7.2 Simulator

The second stage of the pipeline is the **Simulator**, which uses the second list introduced above to iterate through all levels of interest and obtain the corresponding filled vertices. At this stage, virtual humans are considered as individual 3D points, and depending on the ROI, the proper motion planning method is applied. In other words, the Simulator ensures that each virtual human instance comes closer to its next waypoint, *i.e.*, its next short-term goal, and handles dynamic inter-pedestrian collision avoidance. *YaQ* distinguishes three different levels of interest:

Level 0: regions of high interest are typically zones in front of the camera, or where particular events are happening. Such regions are governed by a potential field-based algorithm similar to [Treuille et al., 2006], based on a precomputed grid. To avoid the costly spreading of the potential field, we limit its computation to the sole region of high interest. Note that in regions of this level of interest, pedestrians are steered towards special waypoints corresponding to the center of a neighbor cell with the lowest potential. The navigation graph waypoints are not used here.

Level 1: in regions still visible but of lower interest, pedestrians are smoothly steered towards their navigation graph waypoint with an algorithm similar to Reynolds' seek behaviour [Reynolds, 1999]. In addition, a short-term collision avoidance method is exploited: taking advantage of the grid structure, a pedestrian checks in the cells ahead if another pedestrian is close by. If it is the case, an intermediate waypoint is introduced to avoid the collision. Once the collision is resolved, the pedestrian is assigned its next navigation graph waypoint again, and resumes its progress on its path.

Level 2: in regions of no interest, *i.e.*, outside the camera view frustum, pedestrians are steered linearly towards their next navigation graph waypoint and do not perform any collision avoidance.

The last operation performed by the Simulator is to update the behavior of the virtual humans. The navigation graph vertices contain zero or more semantic keywords corresponding to a specific behavior. For each vertex annotated with a specific behavior, the Simulator applies the corresponding actions to its pedestrians. The resulting behaviors are typically expressed through accessories acquisition and dynamic animation changes, *e.g.*, going out of a shop with a shopping bag, or looking at a specific interest point.

7.3 Animator

The Animator, the third stage of the real-time pipeline, is responsible for the animation of each virtual human, whichever the representation it is using, *i.e.*, deformable mesh, rigid mesh, or impostor. The lists of visible virtual humans, sorted by human template, LOD, and appearance set in the Scaler phase, are the main data structure used in this stage.

The three different rendering levels of detail are depicted in Figure 9.1. The animation process is similar for all representations: depending on the animation time, the correct keyframe is identified and retrieved. Then, each representation is modified accordingly.

Deformable Meshes. Below are described the specific tasks that are achieved for the deformable meshes:

```
For each human template:
   get its skeleton,
   For each deformable mesh LOD:
      For each appearance set:
         For each virtual human id:
            get the corresponding body,
            update the animation time (normalized between 0.0 and 1.0),
            perform general skeletal animation,
            perform facial skeletal animation,
            perform hand skeletal animation.
```

Note that the second loop iterates over several LOD of deformable meshes. This situation happens when several meshes with a different number of triangles are used, *e.g.*, we can use deformable meshes of 6,000 triangles at the fore-front and less detailed deformable meshes, of 1,000 triangles behind. Performing a skeletal animation, whether it is for the face, the hands or all the joints of a virtual human, can be summarized in four steps. First, the correct keyframe, depending on the animation time, is retrieved. Note that at this step, it is possible to perform a blending operation between two animations. The final keyframe used is then the interpolation of the ones retrieved from each animation. The second step is to duplicate the original skeleton relative joint matrices in a cache. Then, in the cache, the matrices of the joints modified by the keyframe are overwritten. Finally, all the relative matrices (including those not overwritten) are multiplied to obtain global matrices, and each of them is post-multiplied by the inversed global matrices of the skeleton. Note that optional animations, like facial animation, are usually performed only for the best deformable mesh LOD, *i.e.*, the most detailed mesh, at the fore-front.

Rigid Meshes. For the rigid meshes, the role of the Animator is much reduced, since all the deformations are pre-computed:

```
For each human template:
   For each rigid mesh LOD:
      For each appearance set:
         For each virtual human id:
            get the corresponding body,
            update the animation time (between 0.0 and 1.0).
```

Note that once again, we could pre-compute rigid animations for several meshes with a different number of triangles. This is why the second loop is introduced. However, most of the time, we limit the number of rigid mesh LOD to 1, for storing animations for several meshes would quickly become too expensive.

Impostors. Finally, for the impostors, since a keyframe of an impostor animation is only represented by two texture atlas, no specific deformation needs to be achieved. However, we assign the Animator a special job: to update a new list of virtual human ids, specifically sorted to allow a fast rendering of impostors. This list was previously introduced as the animation list in Section 7.1.2. At initialization, and for each human template, a special list of virtual human ids is created, sorted by appearance set, impostor animation, and keyframe. The first task achieved by the Animator is to reset the impostor specific list in order to refill it accordingly to the current state of the simulation. To refill this list, an iteration is performed over the current rendering list (sorted by human template, LOD, and appearance set), which has just been updated in the Scaler stage:

```
For each human template:
   get its impostor animations,
   For the only impostor LOD:
      For each appearance set as:
         For each virtual human vh:
            get the body of vh,
            update the animation time (between 0.0 and 1.0),
            get body’s current impostor animation id a,
            get body’s current impostor keyframe id k,
            put vh’s id in special list[as][a][k].
```

This way, the impostor specific list is updated every time the data passes through the Animator stage, and is thus ready to be exploited at the next and last stage, the Renderer.

7.4 Renderer

The Renderer represents the phase where draw calls are carefully issued to the GPU to display the environment and the crowd. Indeed, it is important to minimize first the state changes overhead, and second, the number of draw calls.

7.4.1 Shadows

In our architecture, illumination ambiances are set from four directional lights, whose direction and diffuse and ambient colors are prealably (or interactively) defined by the designer.

The light coming from the sun is the only one casting shadows. As we lack a real-time global illumination system, the three other lights are present to provide enough freedom for the designer to give a realistic look to the scene. This configuration has given us satisfaction as we mainly work on outdoor scenes. See Figure 7.2 for results.

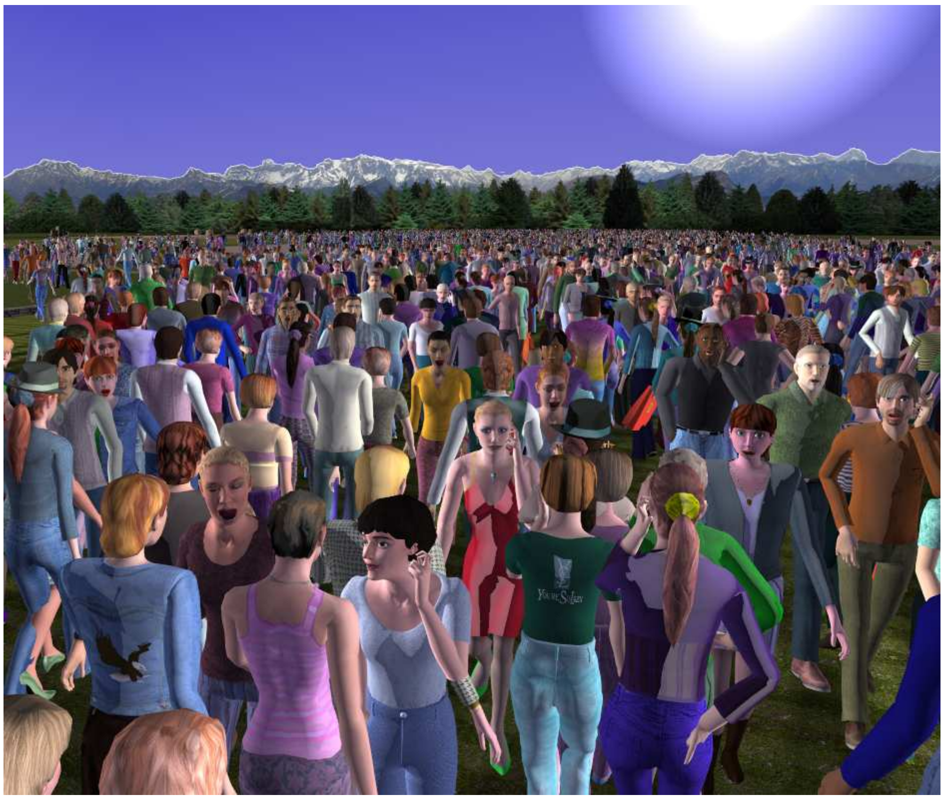


Figure 7.2: Dense crowd in a large environment.

Virtual humans cast shadows on the environment and, reciprocally, the environment casts shadows on them. This is achieved using a shadow mapping algorithm [Williams, 1978; Woo et al., 1990] implemented on the GPU. At each frame, virtual humans are rendered twice:

- The first pass is from the directional light view perspective, *i.e.*, the sun. The resulting z-buffer values are stored in the shadow map.

- The second pass is from the camera view perspective. Each pixel is transformed into light perspective space and its z value is compared with the one stored in the shadow map. Thus, it is possible to know if the current pixel is in shadow or not.

So, we need to render twice the number of virtual humans really present. Though with modern graphics hardware, rendering to a z-only framebuffer is twice as fast as rendering to a complete framebuffer, one expects a certain drop in the frame rate. Moreover, standard shadow mapping suffers from important aliasing artefacts located at shadow borders. Indeed, the resolution of the shadow map is finite, and the larger the scene, the more aliasing artefacts appear. To alleviate this limitation, several strategies are used:

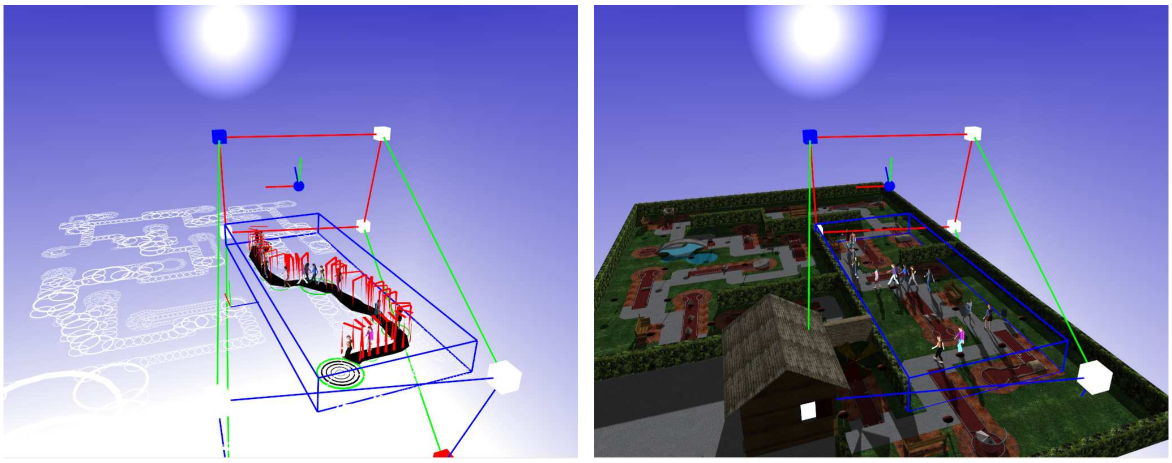


Figure 7.3: Shadowed scene with apparent directional light frustum.

- Dynamically constrain the shadow map resolution to visible characters, and
- Combine percentage closer filtering [Reeves et al., 1987] with stochastic sampling [Cook, 1986], to obtain fake soft shadows [Uralsky, 2005].

We now further describe how to dynamically constrain the shadow map resolution to visible characters. A directional light, as its name indicates, is defined only by a direction. Rendering from a directional light implies using an orthographic projection, *i.e.*, its frustum is a box, as depicted in Figure 7.3. An axis-aligned bounding box (AABB) is a box whose faces have normals that coincide with the world axes [Möller and Haines, 1999]. They are very compact to store; only two extreme points are necessary to determine the whole box. AABB are often used as bounding volumes, *e.g.*, in a first pass of a collision detection algorithm, to efficiently eliminate simple cases.

A directional light necessarily has an orthographic frustum aligned along its own axes. So, we can consider this frustum as an AABB. The idea is to compute at each frame the box englobing all the visible virtual humans, so that it is as tight as possible. Indeed, using an AABB as small as possible allows to have a less stretched shadow map. At each frame, we compute this AABB in a four-step algorithm:

1. The crowd AABB is computed in world coordinates, using visible navigation graph vertices. By default, the AABB height is set to two meters, in order to bound the characters at their full height.
2. The light space axes are defined, based on the light normalized direction L_z:

   L_x = normalize($(0, 1, 0)^T \times L_z$).

   L_y = normalize($L_z \times L_x$).
3. The directional light coordinate system is defined as the $3{\times}3$ matrix $M_l = [L_x, L_y, L_z]$.
4. The eight points composing the AABB (in world coordinates) are multiplied by M_l^{-1}, *i.e.*, the transpose of M_l. This operation expresses these points in our light coordinate system.

Note that remultiplying the obtained points by M_l would express the crowd AABB back into world coordinates. In Figure 7.3 are illustrated the shadows obtained with this algorithm. Practically, to be able to choose an adequate resolution given the situation, *e.g.*, detailed shadows for characters close to the camera, we use three different shadow maps: one for the shadows cast by the environment, one for the people (deformable and rigid meshes) near the camera, and one for people far from it (impostors).

7.4.2 Virtual Human Rendering

Once the first pass has been executed, the second pass is used to render all the virtual humans and their shadows. To reduce state change overhead, the number of draw calls are minimized, thanks to our rendering list of visible humans sorted by human template, LOD and appearance set.

Deformable Meshes. In the following pseudo-code, we show the second pass in the deformable mesh rendering process:

```
For each human template:
   For each deformable mesh LOD:
      bind vertex, normal, index, and texture buffer,
      send to the GPU the joint ids influencing each vertex,
      send to the GPU their corresponding weights,
      For each appearance set:
         send to the GPU  texture specular parameters,
         bind texture and segmentation maps,
         For each virtual human id:
            get the corresponding body,
            send the joint orientations from cache,
            send the joint translations from cache.
```

Note that the process of the first pass is quite similar, although data useless for shadow computation is not sent, *e.g.*, normal and texture parameters. In this rendering phase, one can see the full power of the sorted lists: all the instances of a same deformable mesh have the same vertices, normals and texture coordinates. Thus, these coordinates need to be binded only once per deformable mesh LOD. The same applies for the appearance sets: even though they are used by several virtual humans, each needs to be sent only once to the GPU. Note that each joint transformation is sent to the GPU as two vectors of four floating points, retrieved from the cache filled in the Animator phase.

Rigid Meshes. For the rigid meshes, the process is quite different, since all vertex deformations have been achieved in a pre-process. We develop here the second pass in pseudo-code:

```
For each human template:
   For each rigid mesh LOD:
      bind texture coordinate buffer,
      bind indices buffer,
      For each appearance set:
         send to the GPU texture specular parameters,
         bind texture and segmentation maps,
         For each virtual human id:
            get the corresponding body,
            get the correct rigid animation keyframe,
            bind its vertex and normal buffers.
```

In the rendering phase of the rigid meshes, only the texture coordinates and indices can be binded at the LOD level, in opposition to the deformable meshes, where all mesh data is binded at this level. The reason is obvious: for a deformable mesh, all the components representing its mesh information (vertices, normals, *etc.*) are the same for all instances. It is only later, on the GPU, that the mesh is deformed to fit the skeleton posture of each individual. For a rigid mesh, its texture coordinates, along with its indices (to access the buffers), remain the same for all of their instances. However, since the vertices and normals are displaced in a pre-process and stored in the keyframes of a rigid animation, it is only at the individual level, when we know the animation played, that their binding can be achieved.

Note that since the vertices sent to the GPU are already deformed, there is no specific work to be achieved in the vertex shader. Concerning the shadow computation phase, *i.e.*, the first pass, the pseudo-code is the same, but without sending useless data, like normal and texture information.

Impostors. Rendering impostors is fast, thanks to the animation list, which is sorted by human template, appearance set, animation, and keyframe, and is updated at the Animator phase. Here follows the corresponding pseudo-code:

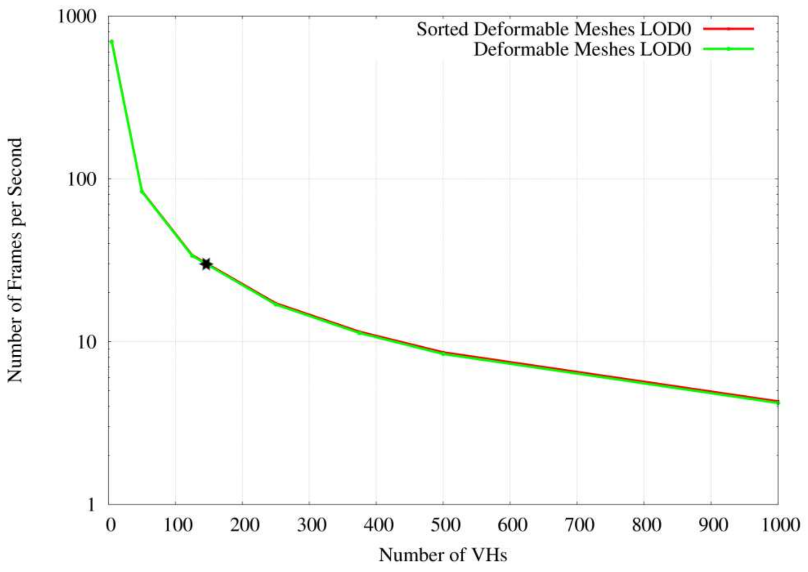
```
For each human template:
   get its impostor animations,
   For each appearance set:
      bind texture and segmentation maps,
      For each impostor animation:
         For each keyframe:
            bind normal map,
            bind UV map,
            For each virtual human id:
               get the corresponding body,
               get the correct point of view,
               send to GPU texture coordinates where
               to get the correct virtual human posture
                         and point of view.
```

With our dedicated list, if several virtual humans issued from the same human template and appearance set are playing exactly the same keyframe, the normal and UV atlases are bound only once.

7.4.3 Performance

In this chapter, we have detailed different steps of *YaQ*'s pipeline to simulate crowds.

We now expose the performance obtained with this architecture. As a reminder, in Figure 3.1, the storage requirements are summarized, depending on the animation types. In Figures 7.4 and 7.5, we compare the frame rates obtained in two cases. Firstly, when sorted virtual human lists are exploited, as detailed in Section 7.1.2. Secondly, when the runtime animation and rendering stages do not use sorted lists, but directly each virtual human, one



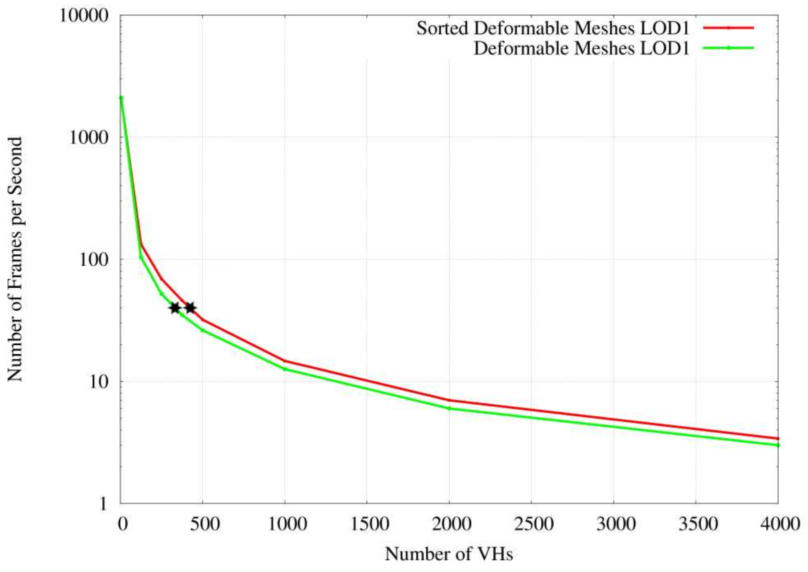


Figure 7.4: Frames per second obtained for (*top*) highly detailed deformable meshes, and (*bottom*) simple deformable meshes. The red lines show the results obtained when working with sorted lists, the green ones with a naive approach. The stars indicate the results for 30 frames per second.

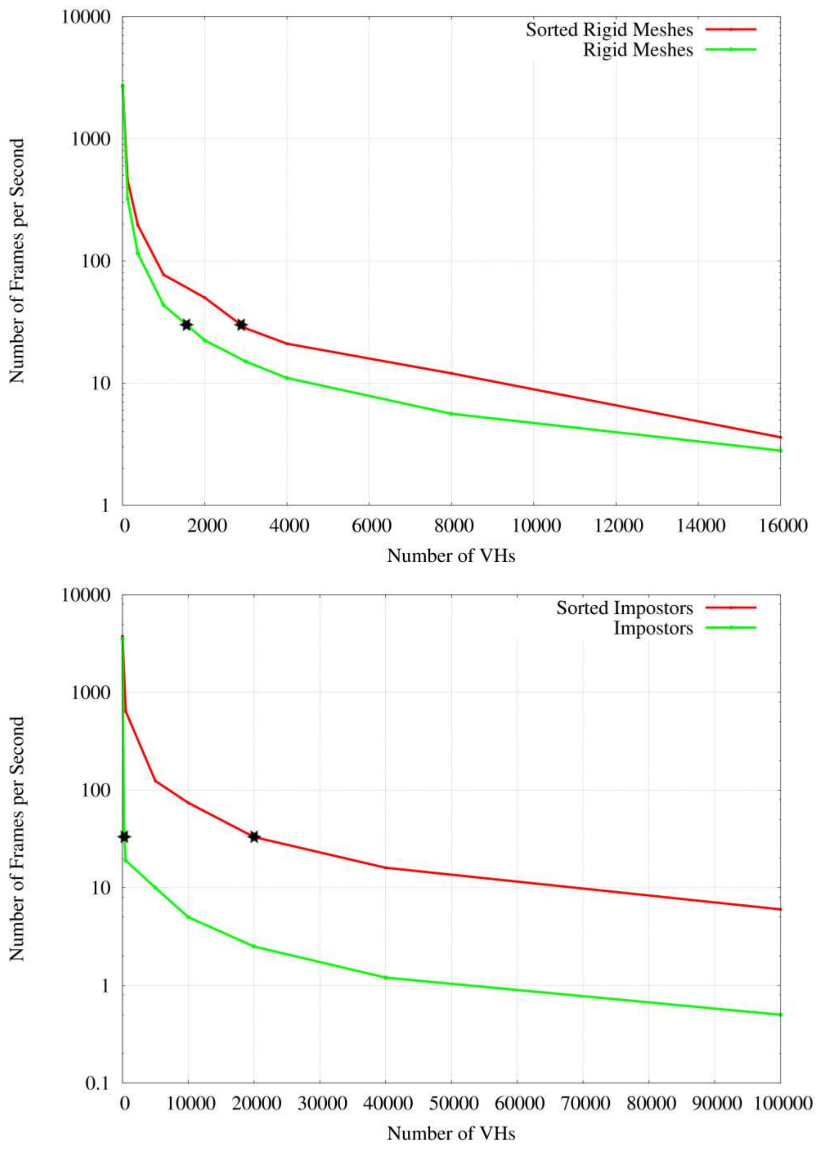


Figure 7.5: Frames per second obtained for (*top*) rigid meshes, and (*bottom*) impostors. The red lines show the results obtained when working with sorted lists, the green ones with a naive approach. The stars indicate the results for 30 frames per second.

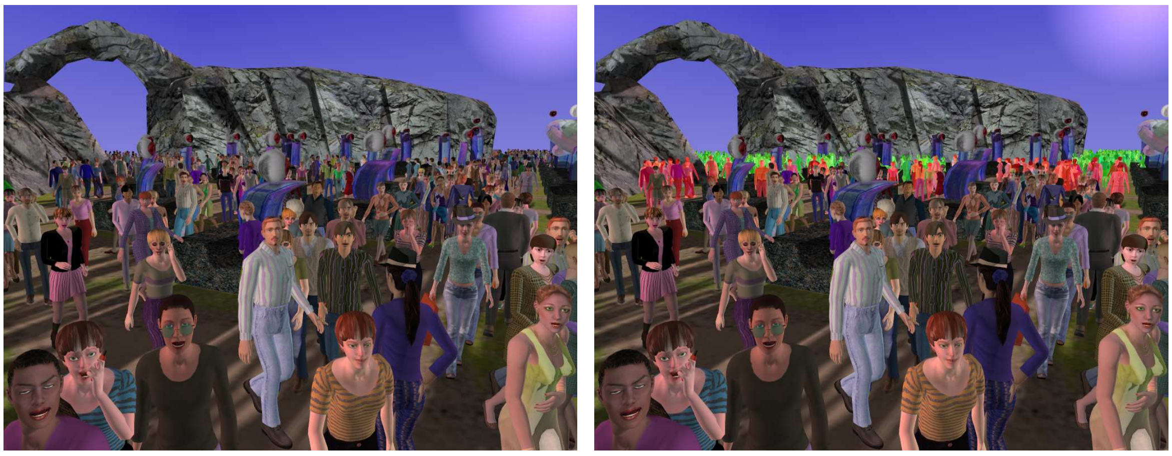


Figure 7.6: (*left*) Virtual humans navigating in a complex environment. (*right*) Same configuration of virtual humans with apparent levels of detail; in red: the rigid meshes, in green: the impostors.

after another, in no specific order. With such a process, all the information needed by the GPU has to be sent for each virtual human, independently from the data that may be shared by several of them. The conditions in which the tests have been achieved are as follows: five human templates, steering and animation enabled, no shadows, no accessories, no collision avoidance. As one can observe in Figure 7.4, when using highly detailed deformable meshes, the results obtained with or without sorted lists are almost similar. This can be explained by the communication sent from the CPU to the GPU (joint transmission): such transmissions imply a pipeline flush for each rendered virtual human, thus becoming the bottleneck of the application. However, when less detailed representations are exploited, the advantage of sorting the lists becomes clear in Figure 7.5. An image directly obtained from our running architecture is shown in Figure 7.6. On the right-side, one can observe the distance at which the virtual humans switch to lower representations: in red are the rigid meshes, and in green the impostors.

CHAPTER 8

Results and Case Studies

In this chapter, we present the results obtained with *YaQ*. We analyze the achieved performance, and present various case studies in which *YaQ* has been successfully used:

- **Case Study 1: Buildings.** In Section 8.1, we illustrate the versatility of segmentation maps (described in Chapter 4) by using them in a completely different context.
- **Case Study 2: Theme Park.** In Section 8.2, we present a concrete example where accessories (Chapter 5) and segmentation maps (Chapter 4) are combined to simulate a theme park populated with human instances that are unique in appearance.
- **Case Study 3: Ancient Pompeii.** In Section 8.3, we present how *YaQ* has been used to simulate a Roman population in the Ancient city of Pompeii. This application has been developed in the framework of a European project on Cultural Heritage.
- **Case Study 4: Medical Domain.** In Section 8.4, we present two applications of *YaQ* in the medical domain. Firstly, *YaQ* is currently used in the office of a physical therapist to treat some patients' equilibrium problems. The second application of *YaQ* aims at supporting the work of psychiatrists in helping victims of agoraphobia (with crowds). This application has not yet been tested on actual patients, because we are finalizing the integration of *YaQ* in a CAVE[1] to improve the feeling of immersion of future patients.

[1]CAVE is the abbreviation for Cave Automatic Virtual Environment. It is a large cube in which a person can stand. Each cube face is a screen on which virtual reality images are projected for the viewpoint of the person, thus drastically improving the feeling of immersion for a user, as compared to a single screen.

8.1 Case Study 1: Buildings

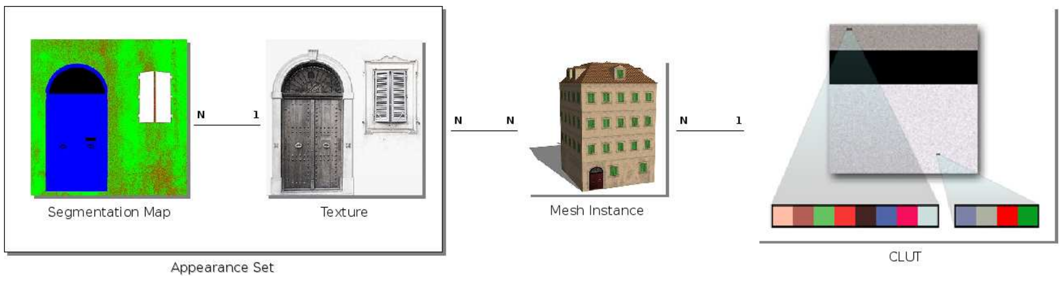


Figure 8.1: Relations between data structures. Each mesh instance possesses a set of colors in the CLUT. Applying colors smoothly to the texture parts is achieved with the use of segmentation maps.

We have fully detailed our approach and implementation of appearances sets in Chapter 4. We have presented images where this approach was successfully applied on virtual humans. In this section, we demonstrate the versatility and robustness of appearance sets by using them on an additional example. Indeed, although the issue of appearance variety may seem limited to crowds, it can be encountered in many other cases, where objects are susceptible to be instantiated several times. The example we present here is the case of buildings, instantiated many times to form a city.

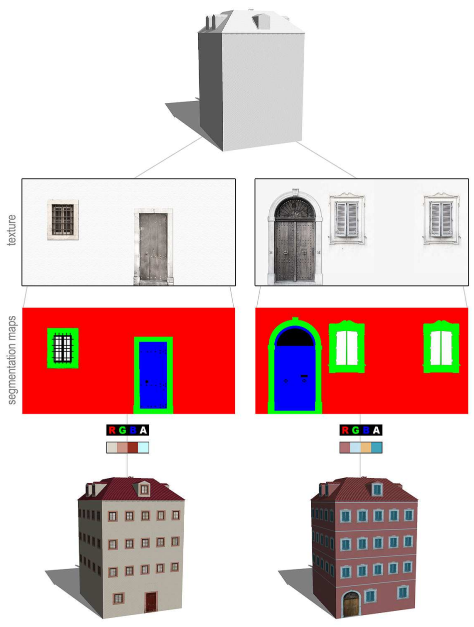


Figure 8.2: Each instance of a building template randomly chooses an appearance set (a texture and a segmentation map).

Applying color variety to buildings in a city is similar to the crowd approach. In Figure 8.1, we describe the different data structures employed. At the creation of a building template, the designer produces a series of appearance sets, each composed of one texture, and one or more segmentation maps, depending on the number of parts composing the building. As a reminder, note that each segmentation map cannot differentiate more than four parts, one per channel. When the building template is instantiated, one color is chosen for each part, and stored in a Color Look-Up Table (see Section 4.3.2).

For each building template, several appearance sets can be created. Figure 8.2 shows some examples of appearance sets applied to the same building template. As a result, a

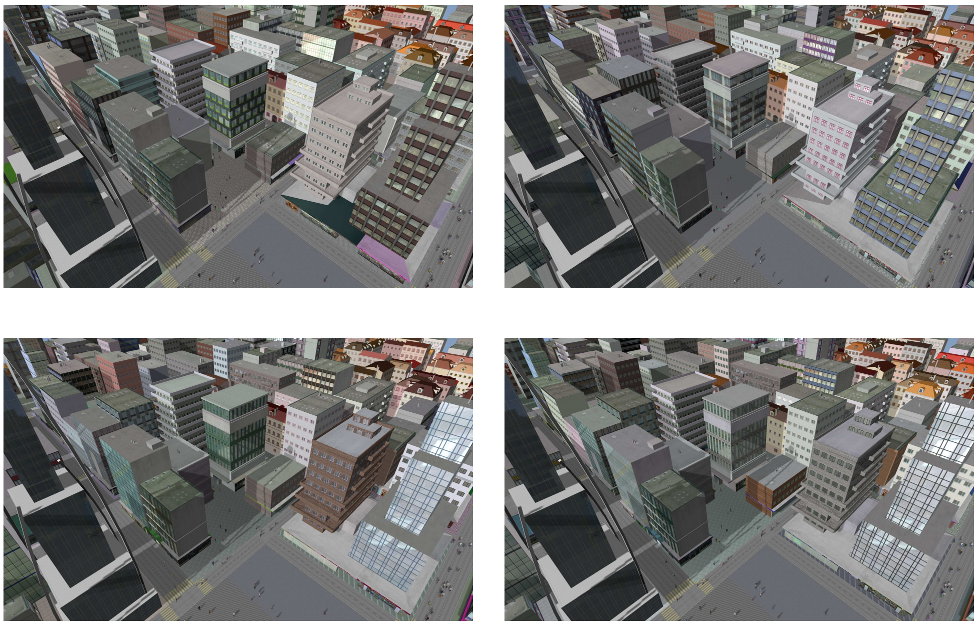


Figure 8.3: Appearance sets applied to instances of six building template.

designer can easily modify building colors, patterns, and material properties. In Figure 8.3, buildings take full advantage of appearance sets: although a single texture is applied to each of them, thanks to the segmentation maps, it is possible to identify the windows, and define particular specular and reflective parameters for them.

8.2 Case Study 2: Theme Park

We have previously detailed various techniques that can be used to vary the appearance of human instances (see in Figure 8.4 a crowd using color variety techniques and accessories).

In this section, we present a case study of these techniques, where appearance sets and accessories have been combined to generate a crowd of unique human instances. To further illustrate the versatility of the segmentation map method, accessories are also segmented so as to increase their variety. Note that we do not further detail these techniques here, for they have already been presented. We rather show the results and performance obtained in the concrete simulation of a large crowd of varied instances in a specific context.

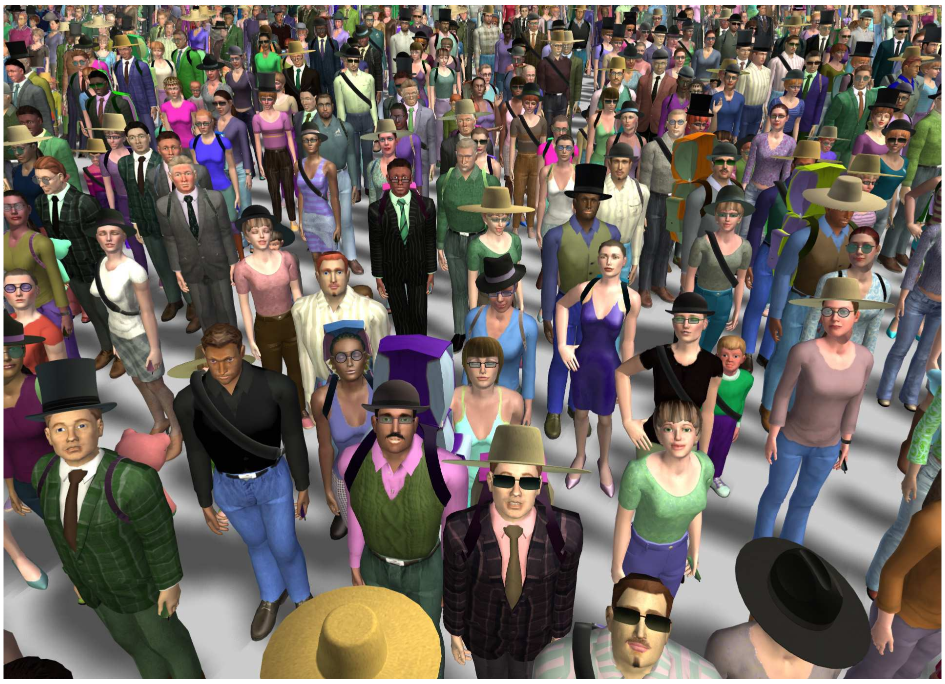


Figure 8.4: Appearance sets composed of two segmentation maps (eight body parts) applied to instances of six human templates. Accessories are varied with appearance sets composed of one segmentation map (maximum four different parts).

8.2.1 Scenario

Our particular scenario takes place in a theme park. The park is composed of numerous doorways that cluster the environment into several atmospheres. When a character passes through a doorway, its colors and accessories change according to the area theme: circus, heaven, hell, halloween, etc. With this scenario, we demonstrate that our appearance variety methods can be integrated to complex real-time crowd applications. A video of this specific simulation is available online [Maïm et al., 2009]: the video shows a simulation of over $5,000$ characters moving in the theme park environment (of $50,000$ triangles). Dynamic shadows are computed for the characters and the scene, navigation and collision avoidance are enabled, and our three rendering LOD are used. Figure 8.5 illustrates crowds benefiting from our techniques in the theme park with two images. With accessories and segmentation maps, we obtain unique and visually appealing individuals.

8.2.2 Performance

The following results and the video introduced above have been produced on an AMD64 X2 5200 with 2GB of RAM and an Nvidia 7900 GTX 512MB graphics board. Virtual humans and accessories are rendered using OpenGL pseudo-instancing.

In Figure 8.6, we show the number of characters, each wearing 0 to 3 accessories, that we can render at 30 fps. For deformable meshes (Figure 8.6 (a)), the number of displayable characters only slightly changes, since the bottleneck of the pipeline remains the mesh skinning.

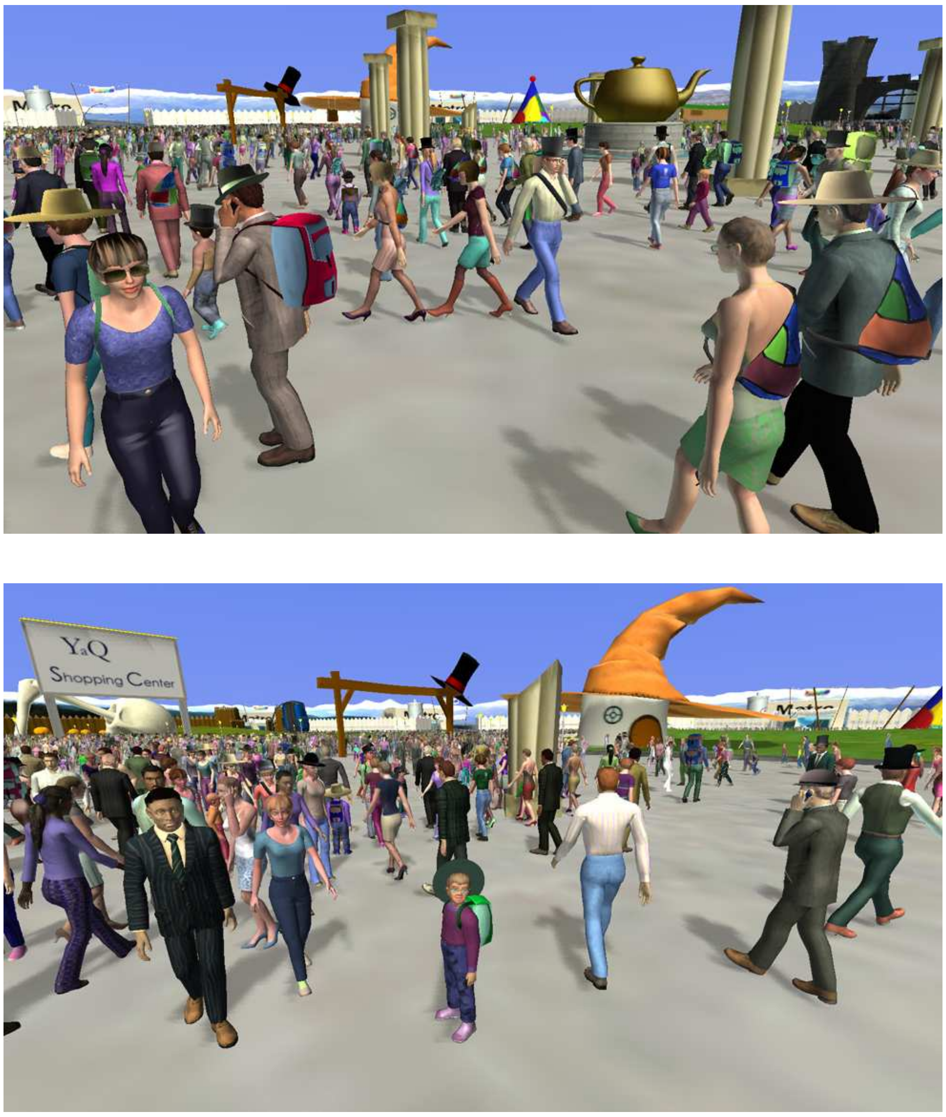


Figure 8.5: Unique characters simulated in a theme park.

Also, each character has over $6,000$ triangles, *i.e.*, 5 to 12 times an accessory triangle count. With rigid meshes (Figure 8.6 (b)), accessories are proportionally more costly, for there is no skinning phase, and the cost of rendering an accessory triangle sums up to the same as rendering a rigid mesh triangle. As for impostors (Figure 8.6 (c)), displaying an accessory or a character has the same fixed cost: two triangles. The loss of performance is thus sizable. However, it is possible in several cases to avoid rendering accessory impostors that are sufficiently small to be indistinguishable at far distances, *e.g.*, glasses, jewelry. In this manner, many more human impostors can be rendered than depicted in Figure 8.6 (c). The fragment depth computation implies disabling early culling optimizations done by the GPU. However, as shown in Figure 8.6 (d), the more accessory impostors, the less this computation affects the frame rate. Figure 8.6 also depicts results with and without segmentation maps. For all LOD, using segmentation maps implies less characters, because of the additional pixel color computation. However, this cost is not prohibitive.

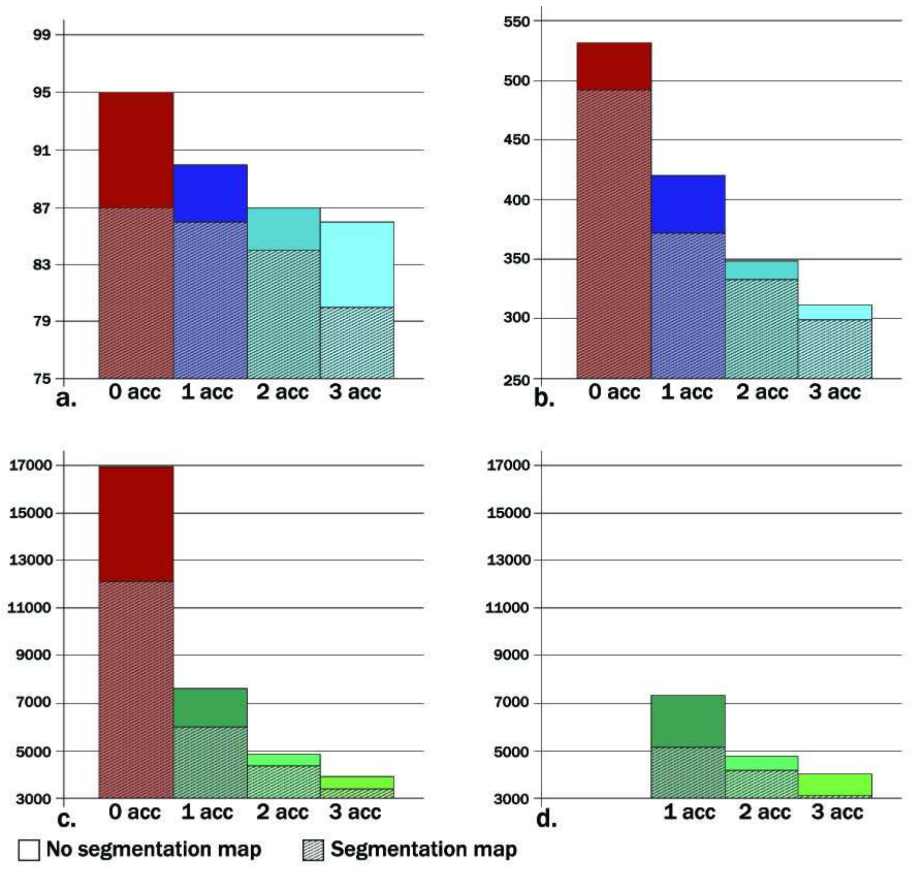


Figure 8.6: Number of displayable characters at 30 fps with 0 to 3 accessories for *(a)* deformable meshes, *(b)* rigid meshes, *(c)* impostors, and *(d)* impostors with depth computation.

Finally, with no accessories nor segmentation maps, larger crowds can be rendered, but they are composed of numerous similar instances, which is not desirable. With our methods, instead of huge crowds of mirror image characters, we offer large crowds of unique instances. In Figure 1.2, only five human templates are instantiated several times, fully exploiting their textures, accessories, and segmentation maps. To illustrate the versatility of the segmentation map method, accessories are also segmented to increase their variety. We demonstrate in the mentioned video that our methods can be integrated to complex real-time crowd applications.

8.3 Case Study 3: Ancient Pompeii

The architecture of *YaQ* has also been used to simulate crowds in a real-time cultural heritage application: Pompeii was a Roman city, destroyed and completely buried during an eruption of the volcano Mount Vesuvius. We have revived its past by populating a 3D model of its previous appearance with crowds of Virtual Romans. In this section, we introduce how *YaQ* has been able to simulate Ancient Pompeii life in real time.

First, we receive in input an annotated city model, generated using procedural modeling [Müller et al., 2006; Maïm et al., 2007]. These annotations contain semantic data, such as land usage, building age, and window/door labels. *YaQ* is responsible for automatically interpreting the semantics to populate the environment and trigger special behaviors in the crowd, depending on the location of the characters. A video available online [Maïm et al., 2007] shows the results obtained while simulating virtual Romans in a district of Ancient Pompeii. We do not further detail how the virtual Romans express intelligent behaviors, for it is beyond the scope of this document. We however focus on the visual results, *i.e.*, which variety techniques have been used and how they have been exploited.

Color Variety. For this simulation, seven roman templates were used (two nobles, two plebeians, two patricians, and one legionary). To ensure a varied crowd, each template used segmentation maps to introduce color variety to their clothes, skin, hair, and eyes. Designing the constrained color ranges for each body part (see Section 4.2.2) of each template has been a particularly demanding task; indeed, color spaces have been chosen according to archaeological data: slaves wore dark colors, mainly in shades of brown, while rich Romans exhibited bright colors, as depicted in Figure 8.7.

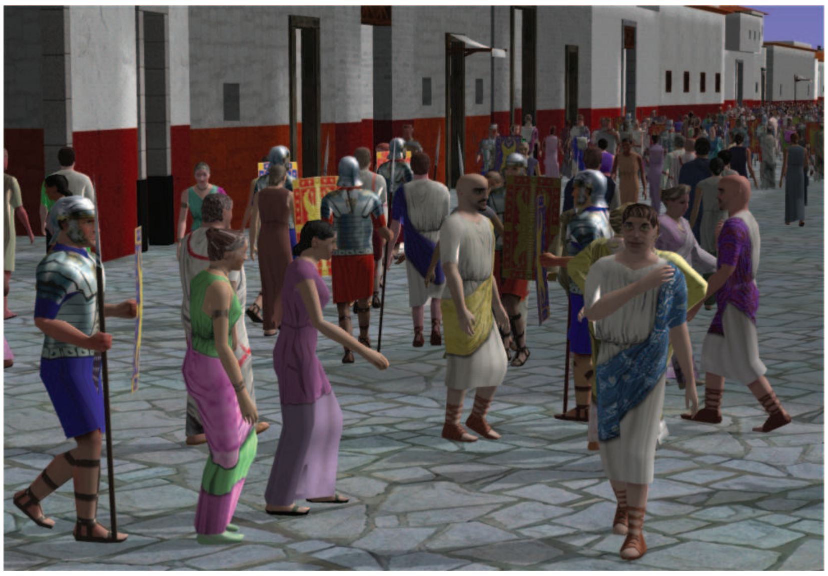


Figure 8.7: Virtual Romans simulated in a reconstructed district of Ancient Pompeii. The colors worn by instances reflect their status: slaves, poor, rich, etc.

Accessories. In order to further increase their appearance variety, the Virtual Romans wear accessories adapted to their time: amphoras and roman bread mainly. To be true to the ancient traditions, amphoras are attached to different joints depending on the Roman's status: middle- and low-class Romans may carry amphoras on their head, while richer instances do not have such practices (see Figure 8.8).

Animation Variety. In the Pompeii simulation, animation has been varied with two techniques: first of all, dedicated upper-body movements are pre-computed and introduced in locomotion animations, as introduced in Section 6.3. These upper body movements have

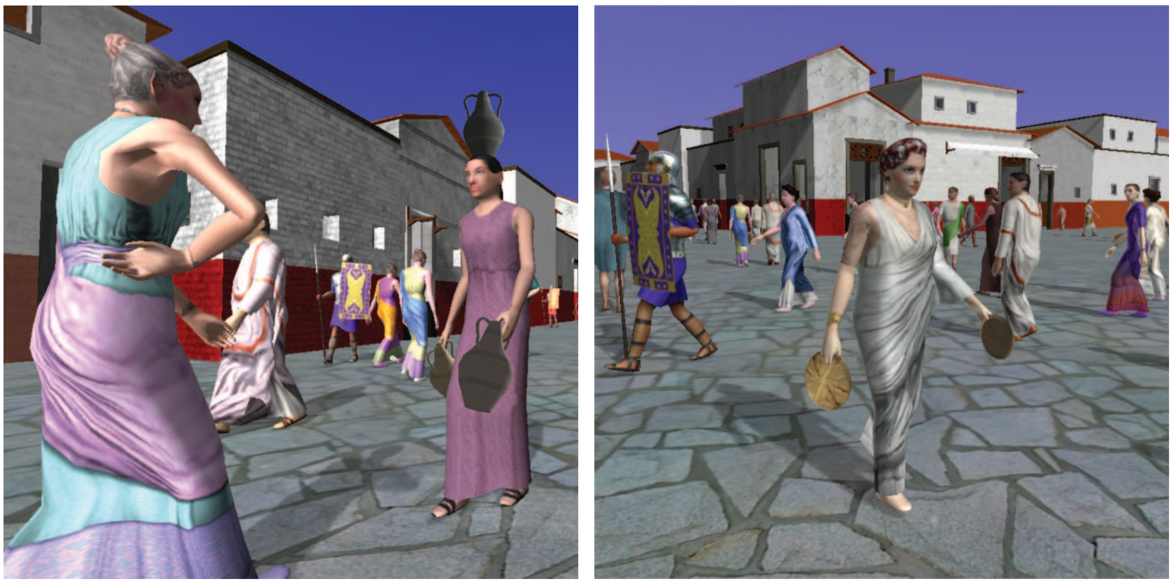


Figure 8.8: (*left*) A middle-class Virtual Roman carries amphoras in her hands and also on her head, while (*right*) a rich Virtual Roman may carry bread in her hands, but would not bear anything on her head.

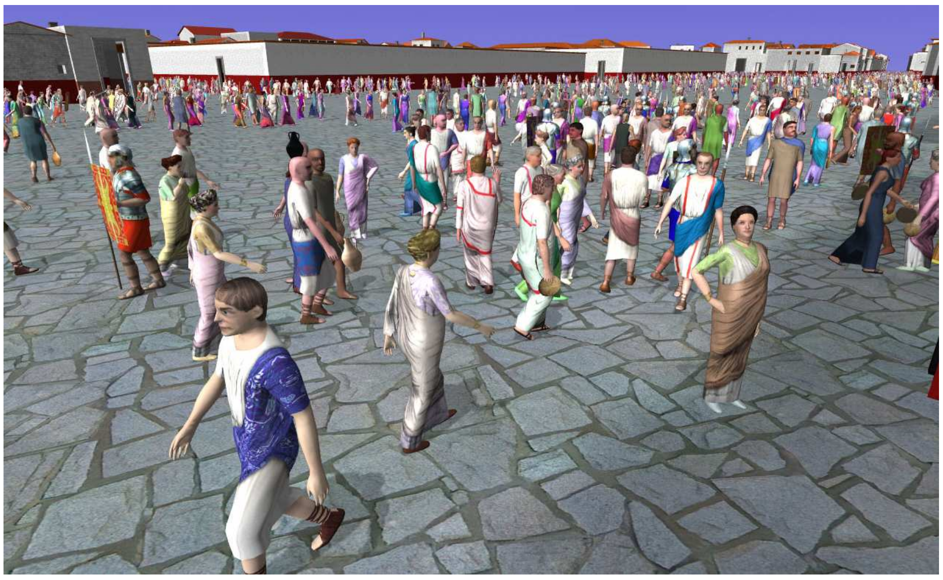


Figure 8.9: Virtual Romans walking in a street of Ancient Pompeii.

been designed specifically for these Roman templates: women can put either hand on their hips (Figure 8.8 (left)); rich men may show their open right palm in a salute position; middle-class men can have their right hand rest on their left shoulder to present a respectful salute (Figure 8.9). The second technique we use to apply animation variety is similar to the method introduced in Section 6.4 to bear complex accessories: freezing some joints in a predefined orientation. In this particular case, we do not freeze joints to constrain the movements of Romans wearing complex accessories, but to bend the back of Roman slaves. This effect is clearly visible on the video referenced above.

8.4 Case Study 4: Medical Domain

As presented in the previous sections, there are many situations and events that can be simulated with *YaQ*. Recently, our crowd engine has aroused the interest of professionals in various fields of the medical domain, to help training patients suffering from diverse pathologies. In Section 8.4.1, we first present how the crowd engine has been used to train patients suffering from equilibrium problems. Then, in Section 8.4.2, we introduce how *YaQ* has been adapted to be used in a CAVE. A previous collaboration with Dr. Riquier, psychiatrist, has been conducted to help patients suffering from social phobia. The idea of integrating crowds in a CAVE could potentially lead to another future collaboration, in order to train patients who are victims of agoraphobia (with crowds) in a virtual environment.

8.4.1 Vestibular Reeducation

Vestibular reeducation is a physical treatment of vestibular instability. This treatment, supervised by a physical therapist, is a balance training that the patient performs, resulting in an improved balance and a decreased risk of falling. Jérôme Grapinet, physical therapist specialized in dizziness and instability since 1989, has collaborated with the EPFL-VRLAB to test a new training exercise for patients with vestibular problems: immerse them in a crowd of virtual humans passing by him (see Figure 8.10).

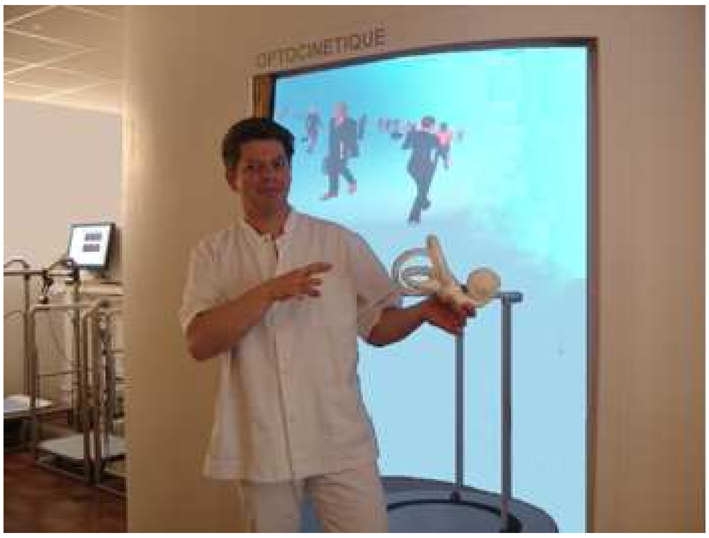


Figure 8.10: Jérôme Grapinet, physical therapist, has created a new exercise for his patients, using *YaQ*.

Mr. Grapinet states that the *YaQ* crowd engine answers to a need that has been present for a long time: being able to immerse victims of an inner ear vestibule injury, or victims of instability, in a virtual situation for training purposes. In both cases, *YaQ* can be used to reaccustom vision to random optokinetic stimulations[2] that are unpredictable, and that mimics a real situation which usually greatly troubles the patient: mingling in a crowd, walking in a supermarket, etc.

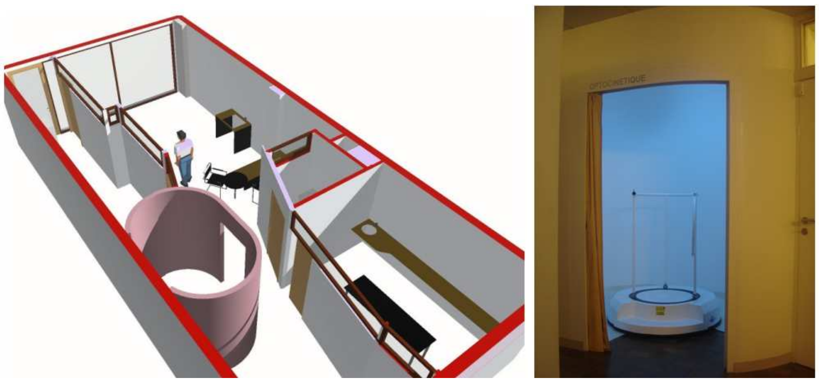


Figure 8.11: *(left)* Mr. Grapinet's projection cabin (in pink) has no right angles, and allows a patient to feel fully immersed in the projected virtual world. *(right)* The projection cabin used to display the crowd simulation of *YaQ*.

Technically, previous attempts to immerse patients in such a manner have been achieved with head mounted displays (HMD). These attempts were not conclusive, for the field of view is too limited with such displays. *YaQ* has been tested in another context, using a projection cabin, especially created to improve optokinetic stimulations (see pink cabin in Figure 8.11 left). This cabin of approximately 6m^2 allows to have a wide field of view with a dome-shaped roof and no border seams, as compared to a CAVE (see Figure 8.11 right).

Mr. Grapinet is satisfied of *YaQ* and uses it to train his patients. Results of a dedicated study he performed will be published in the yearly congress of the International Vestibular Rehabilitation Society (Congrès de la Société Internationale de Réhabilitation vestibulaire), taking place in Luxembourg on May 15th, 2009. More details on this collaboration can also be found on Mr. Grapinet's web site [Grapinet, 2008].

8.4.2 CAVE

We have adapted *YaQ* so that it can be displayed on the multiple screens of a CAVE. Our goal is to help psychiatrists to train their patients victims of agoraphobia (with crowds), by immersing them in a virtual crowd. The CAVE is a very useful installation here, for it almost completely surrounds a user, and thus increases his feeling of immersion. Note that as of today, no actual patient has tested *YaQ* in the CAVE. We first want to test our installation on non-phobic persons to assess the improved feeling of immersion, as compared to using a single projection screen. In this section, we present how *YaQ* has been adapted to simulate crowds in the CAVE with stereo display and head tracking, as depicted in Figure 8.12.

Rendering on multiple screens. The first step to use *YaQ* in the CAVE is to be able to adapt it for rendering on four screens at once. For this process we have used an nVidia

[2]The term "optokinetic" refers to the twitching movements of the eye when moving objects are viewed.

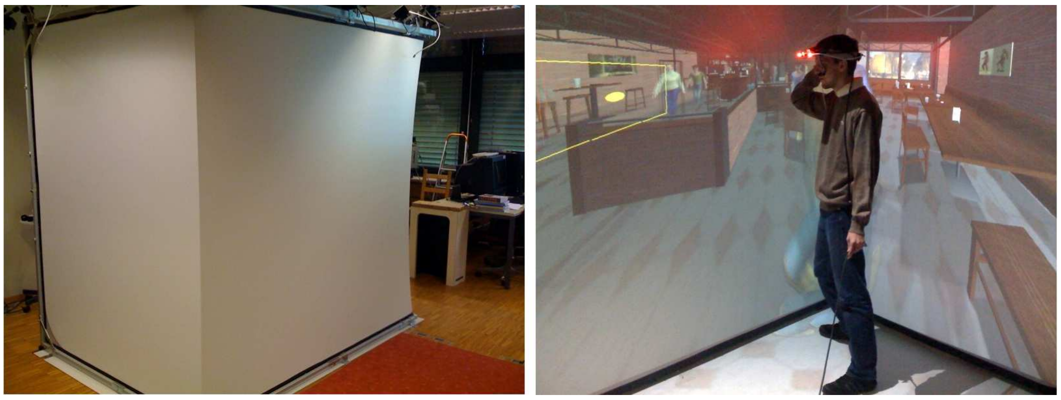


Figure 8.12: The CAVE of the EPFL-VRLAB uses simple home video projectors to immerse a user in a virtual environment. Head tracking enables tracking a user's orientation and position inside the CAVE allowing for interaction with the walking virtual humans.

Quadro Plex system. This system contains 2 GPUs of type Quadro FX 4500 X2, and possesses 8 DVI outputs. A virtual screen is segmented into four different viewports. Each of them represents a single screen in the CAVE. To render an image on all four screens, at each frame, the camera is first positioned to face the front view, then it sequentially also renders each lateral view, and finally the floor view. The process of rendering a single image in the CAVE is thus a four pass rendering operation.

Head tracking. To track the head movements of a user, we use a PhaseSpace motion capture system composed of 8 cameras that surround the screens of the CAVE. The user entering the CAVE wears a head-tracking device composed of three markers that allow the system to track the head position and orientation. Indeed, these informations are required to compute the correct perspective projection for each screen of the CAVE. Whenever the user is moving inside the CAVE, the projection matrices are dynamically modified to sustain the immersion.

Anaglyphic stereo. To further improve the user's immersion level, we achieve anaglyphic stereo rendering using red-and-blue glasses. To do so, we render each frame twice, once for the red channel (left eye) and once for the green and blue channels (right eye).

Screen calibrations. The screens of the CAVE are slightly curved. In order to correct this, each screen image is first rendered to a texture. Then, the texture is applied to a polygon mesh, which is reshaped to match the curvature of the screens.

CHAPTER 9

Conclusion

In this thesis, we have described how the architecture of *YaQ* has been built to ensure the simulation, animation and rendering of large and interactive crowds in real time. We illustrate in Figures 1.1 and 9.1 (top) the results typically obtained when simulating a crowd in an urban environment with *YaQ*, taking advantage of all the variety techniques described before. We distinguish two major contributions in our work. Our first contribution is the general architecture of *YaQ*, its structure, and mechanism to make real-time simulation of large crowds possible. Our second contribution is the introduction of variety at many different levels: color, shape, and animation.

9.1 *YaQ* Architecture

Contributions. The specificity of *YaQ* is to be able to simultaneously address crowd navigation, animation and rendering in real time. *YaQ* integrates techniques at the level of the state of the art for solving each of these problems. Structures dedicated to crowds have been specifically created: human instances are wisely sorted into lists at every frame of the simulation to minimize state switches, motion kits are created to help the animation of individuals at all LOD, a dedicated database is used to intelligently store serialized data that can be shared by several entities. At all steps, *YaQ* allows to set-up a desired trade-off between performance and realism. It can thus be used for various application domains, such as video games where performance is preponderant, to Virtual Reality applications where the level of realism that can be obtained is a crucial criterion.

Future Work. The simulation part of *YaQ* has been designed from scratch to run in a sequential way. Nowadays, there are new opportunities to improve this design by exploiting multi-core CPUs. It would be an interesting topic of research to identify the parts of the simulation that can be parallelized and/or distributed on several cores. For instance, we are confident that major parts of the collision avoidance could be optimized with parallelization. Also, every operation that is achieved on virtual humans individually before rendering could potentially be parallelized. A first step would be to better exploit the several cores of today's home computers. In a second step, distributing the simulation on several machines at the same time would offer the possibility to simulate even larger crowds.

9.2 Appearance Variety

Contributions. To modify the appearance of individuals, we have introduced variety at several levels: for each human template, several textures are created. Then, body parts are identified for each texture, *e.g.*, hair, skin, shirt, skirt, using segmentation maps. We empirically exploit two segmentation maps per texture, and can thus delimit eight body parts, but there is no limitation to this number, and more segmentation maps can be used if more parts need to be identified. Once segmentation maps are designed, each time a virtual human is instantiated, its body part colors are randomly selected within constrained ranges. The sets of chosen colors are then contiguously stored in a Color Look-Up Table on the GPU. Finally, specific materials and fabrics can be simulated by changing a body part's illumination parameters. For instance, it is possible to simulate shiny shoes, glossy lips, leather trousers, *etc*. Thanks to this technique, crowds are enhanced with subtle make-up, freckles and beard effects, or detailed cloth patterns, and smooth transitions between body parts are ensured. We present an example of color variety applied to a single human template in Figure 9.2.

We have presented the different steps of the technique, and shown its versatility by applying it to various examples (humans, accessories, buildings). Finally, variety is scalable, so that it can be applied to all LOD used in *YaQ*, as illustrated Figure 9.1, for instance.

Future Work. In our work, we have prsesented how to modify specular parameters to simulate various types of clothes. The diffuse element of the Phong lighting equation could also be parametrizable by the designer for more variety, or we could completely replace our lighting equation with a more sophisticated one, allowing to simulate materials like velvet, silk, satin, *etc*. More globally, an interesting path we would like to take in the future is to use a single human template, and make all its elements customizable in order to obtain varied crowds: the main idea is to procedurally generate the human mesh at runtime, on the GPU, based on many individual parameters for each body part. Such an approach would need to be scalable, *i.e.*, its structure needs to be thoroughly studied in order to be usable on all LOD.

9.3 Shape Variety

Contributions. Several techniques to modify the shape of a human instance have been proposed. Firstly, the use of accessories much improves the differences at the shape level: elements such as hats, wigs, backpacks are attached to one joint of the human skeleton, and follow its movements when deformed. Two human instances issued from a same human template can thus have different hair cuts for instance, which greatly modifies the perception of a user. Accessories have been studied to be scalable: they can be applied to deformable meshes, rigid meshes, and impostors. A second contribution we have proposed is to modify an instance's shape by working on its skeleton height, and on its mesh width. The resulting effects are impressive, allowing a designer to easily model pregnancy, starting with a flat belly, or making fat bodies from skinny models, *etc.* The main limitation of this second approach is its lack of scalability. However, this is mainly due to a lack of time, rather than an actual deadlock, and solutions to deal with this limitation are proposed in the next paragraph.

Future Work. First of all, the accessories could easily be adapted to skeleton variations, *i.e.*, modifications on bones' height and width. With shape variations at the mesh level however, accessories need to be adapted by hand for each deformed human mesh. Part of our future work consists in searching a way for accessories to automatically adapt to the shape modifications. Second, shape and height variety is a recently studied technique that still requires to be scaled to all LOD. Our current approach can only be applied to deformable meshes. However, we are confident that adaptations can be found to make this new technique scalable. For rigid meshes, the modifications of a skeleton should be saved and applied to its influenced vertices, although this implies a more costly update of rigid meshes. Another solution could be to directly apply a general scaling of the mesh so that the virtual human has the correct height. However, this would require testing to see if this distortion is noticeable or not. As for impostors, the same general scaling of the quad could be applied. Since they are rendered at far distances, the slight stretching of the texture would not be noticeable. Another lead for future work would be to implement the polypostors presented in [Kavan et al., 2008], and adapt directly their 2D polygons to best fit the shape of deformable meshes.

9.4 Animation Variety.

Contributions. Variety at the animation level has been introduced in three ways. Firstly, using a dedicated locomotion engine [Glardon et al., 2004a,b], we are able to generate many walk and run cycles at various speeds, and with different gaits. Secondly, in a pre-process, it is possible to iterate a second time over the generated cycles in order to add upper-body variations that require the use of an IK solver. For instance, this technique allows us to have a walking human with a hand in its pocket, or on its hip without interpenetrating body parts. Last, but not least, we integrate additional upper-body modifications directly at runtime, to allow virtual humans to wear complex accessories, such as shopping bags, suitcases, balloons, *etc.* An example of such accessories is shown in Figure 9.3. The only prerequisite for this approach is that the animation modifications do not cause interpenetration of body parts. At runtime, we simply constrain the joints by freezing their orientation at a given angle, or clamping this angle within a constrained range. Another animation modification that can be achieved at runtime is to perform interactive facial animations. For instance, pedestrians close to the camera can look at the user.

Future Work. We see three main improvements on the animation side. First, height modifications currently cause some problems with the body animation. Indeed, to correctly animate a human instance that has a skeleton of 30 more centimeters for instance, requires a retargeting of the motion clips, which is not yet automatically achieved. To make height modifications compliant with animation, we should adapt our animation database so that motion cycles are not linked to a single template, but sorted according to a leg height. This is perfectly possible, using the locomotion engine previously described. Second, when virtual humans bump into each other they currently slide away from each other resulting in cumbersome foot sliding artefacts. This could be improved by adding collision animation clips in our database. Then at runtime, depending on the type of collision (frontal, lateral, *etc.*), the most appropriate clip would be selected and played. Finally, we would like to enhance our animation repertoire with virtual humans able to climb stairs, sit on and stand up from benches and chairs, picking up objects in the scene, *etc.* This would require the use of a LOD-based real-time IK solver in the system: costly and accurate iterative IK would be used for characters at the forefront while faster analytic solutions would be exploited at farther distances.

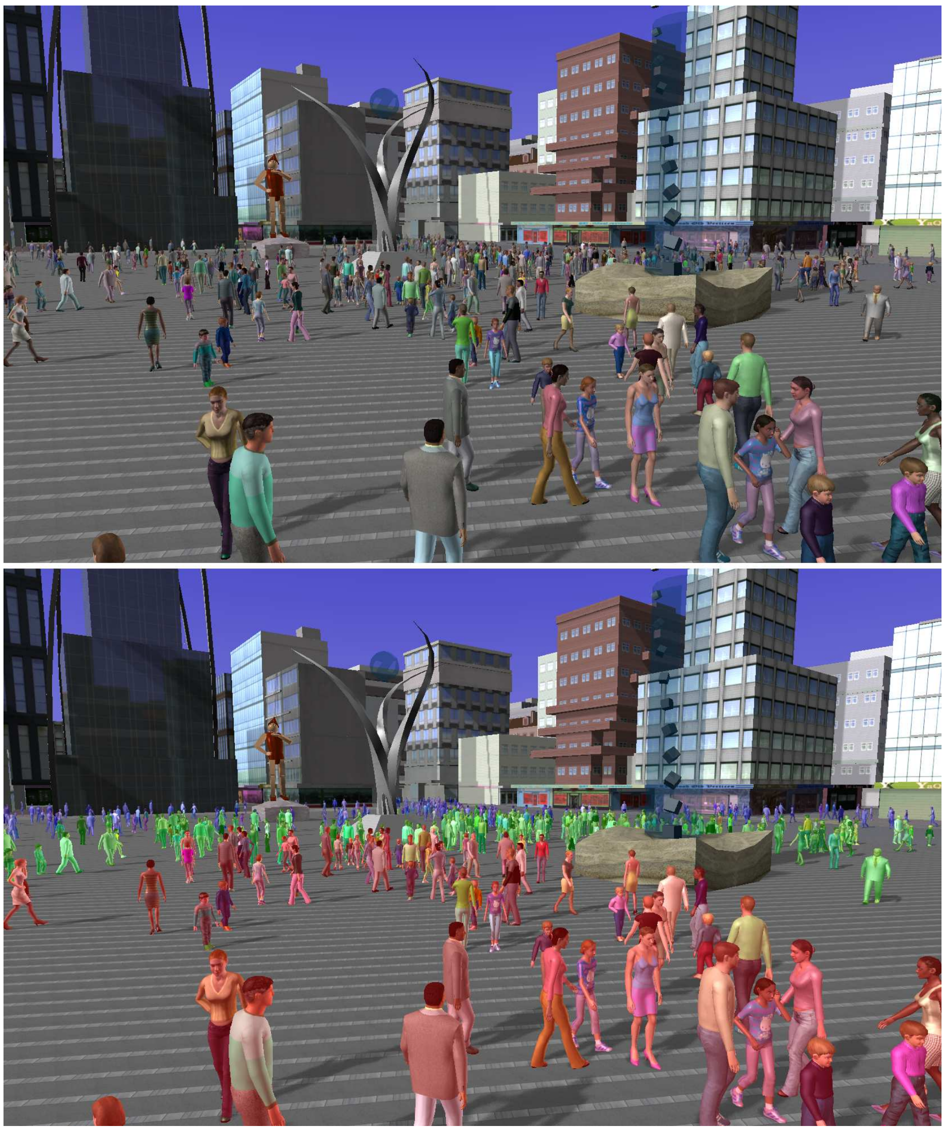


Figure 9.1: *(top) YaQ* simulating and navigating a crowd of varied virtual humans in a city environment. *(bottom)* The 3 different rendering representations become apparent: deformable meshes (in red), rigid meshes (in green) and impostors (in blue).

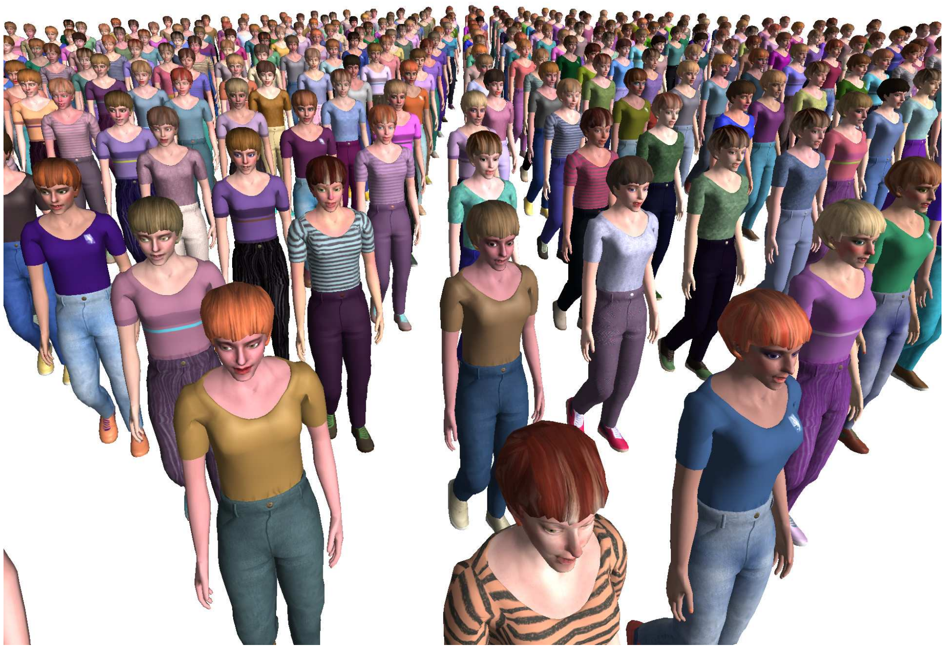


Figure 9.2: Appearance sets applied to instances of a single human template. Note the different specular effects on the body parts and the varying cloth patterns.

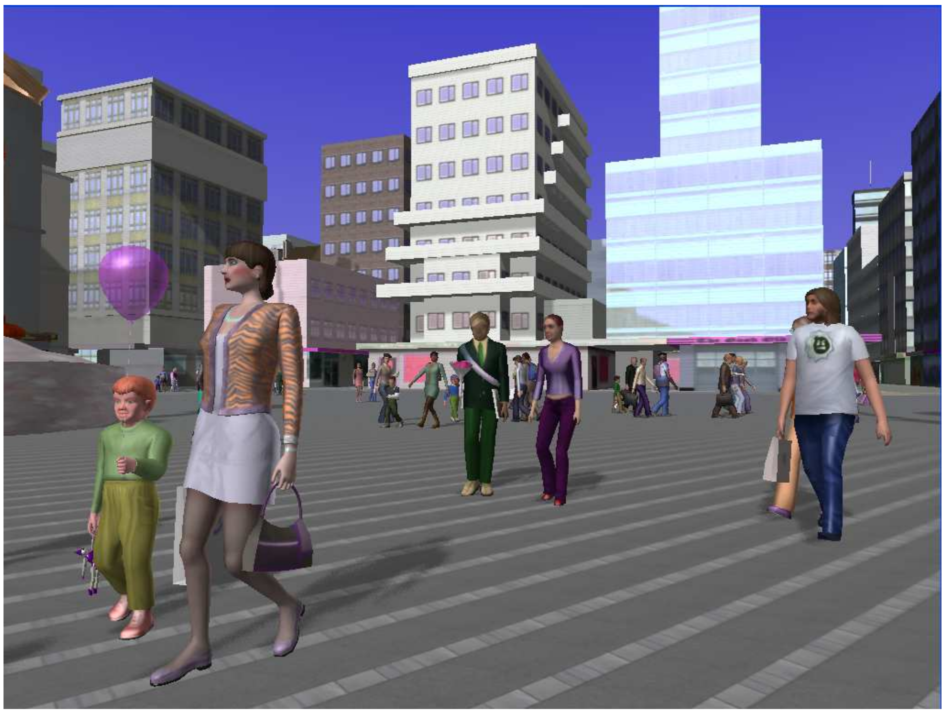


Figure 9.3: Virtual humans can carry complex accessories, such as handbags and balloons. This effect is achieved by freezing and clamping specific joints of the upper body at runtime.

CHAPTER 10

Appendix

10.1 Introduction on Navigation in *YaQ*

YaQ allows autonomous, goal-oriented navigation for pedestrians composing virtual populations. Users control the pedestrians' activity by giving them goals, and consequently, can dispatch the population as desired in a virtual environment. In order to achieve goal-directed navigation, we first need to plan several paths. As illustrated in Figure 3.1, *YaQ* integrates an efficient navigation planning solution based on a navigation graph, which captures the geometry and topology of an environment into a compact data structure [Pettré et al., 2006; Pettré et al., 2007]. A fully detailed presentation of the motion planning techniques used in *YaQ* is available in [Yersin, 2009]

10.1.1 Navigation Graph and Semantic Model

As illustrated in red in Figure 3.1, given a *scene model*, it is possible to compute a *navigation graph*, based on an approximate cell-decomposition of the navigable space. Each cell has a circular shape, and captures a portion of the obstacle-free areas in the environment. Two cells are interconnected when their respective shapes are overlapping. Cells' connectivity is also captured into the navigation graph. This approach is capable of handling complex and large environments, even when composed of uneven surfaces (outdoor terrains) or layered ones (buildings, cities). An example of a navigation graph is displayed in Figure 10.1.

An optional *semantic model* can also be used as a second input to generate a navigation graph annotated with high-level information. Such a model is usually a simplified environment geometry demarcating zones where specific behaviors are expected from pedestrians.

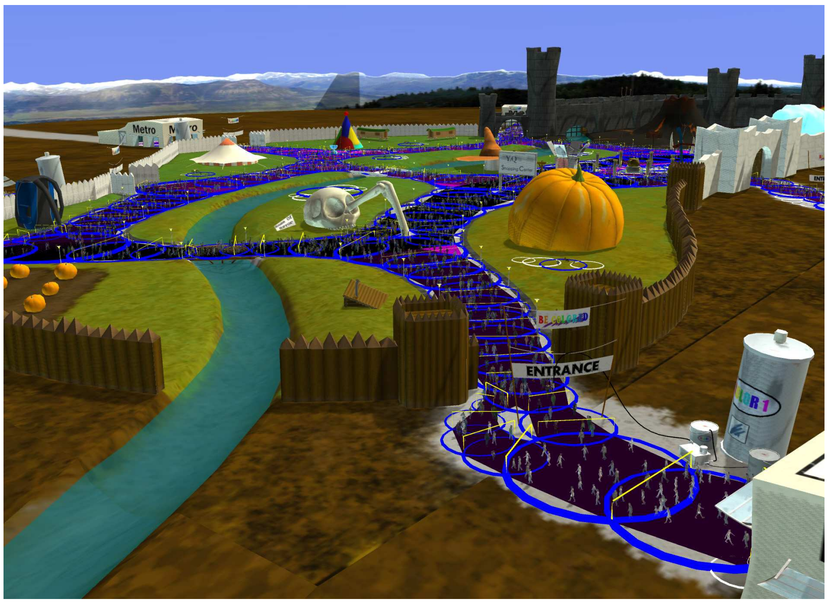


Figure 10.1: A navigation graph is composed of circles representing navigable areas. When two circles intersect, pedestrians are allowed to navigate from one navigable area to the other.

For instance, a semantic model could provide positions of show windows, so that pedestrians look at these points when passing nearby, or areas where shops are situated so that individuals are provided with shopping bags before leaving the shop. Technically, such areas are identified in the semantic model, and all graph vertices enclosed within this area are tagged with a specific behavior. When a pedestrian enters this zone, it will perform the associated action.

10.1.2 Navigation Planning

Once the navigation graph has been generated, it is possible to plan the paths that pedestrians will use. On the one hand, getting a unique and differentiable locomotion trajectory for each pedestrian is essential for preserving believability. On the other hand, attributing a goal to every single pedestrian would result in a fastidious task and expensive computations at run-time. Our *planner* (see Figure 3.1) allows batched processing: first of all, users provide *directives* to choose how the pedestrians will be dispatched in the whole scene, as schematized. A directive simply specifies an initial and a destination point. Secondly, for each directive, the navigation graph is explored using Dijkstra's algorithm in order to find global paths, starting from the shortest one to progressively longer ones. These paths are large passages consisting of navigable areas to cross. As a result, each pedestrian is assigned to one of these paths, giving a first level of variety in trajectories. In a last step, individual trajectories are derived for each pedestrian by selecting singular *waypoints* contained within the global path, resulting in a second level of variety for trajectories. Consequently, each pedestrian moves along its own unique path. Users can create as many directives as desired to populate environments. Note that this individualization of trajectories is achieved only once at initialization and provides realistic results, widely spreading pedestrians having the same destination.

In real time, virtual humans are steered along their planned path by tracking successively waypoints as planned at pre-process. As previously mentioned, different algorithms are used for crowd steering, depending on the environment's regions of interest. For regions of high interest, a costly but accurate potential field-based technique is exploited [Treuille et al., 2006]. To allow the potential field computation at runtime, an additional structure has first to be created at initialization: a *grid*, composed of cells, is disposed over the navigation graph [Yersin et al., 2008].

The resulting locomotion trajectories are unique. However, they also emphasize an individual behavior of pedestrians, *i.e.*, they always walk alone. In any urban environment, it is common to observe people walking in groups of 2 or more. Thus, we have extended the planner to perform the optional task of joining pedestrians in small groups of 2 to 5 people: based on specific user directives, *i.e.*, the number and size of groups within the crowd, the planner takes care of assigning close waypoints on a same path for members of a same group. Note that groups are created randomly with the constraint that the same human template is not used twice in the same group. This extension has proven to strongly impact on the spectator perception of the crowd. The resulting distribution of pedestrians on varied paths and in different groups is illustrated in Figure 10.2.

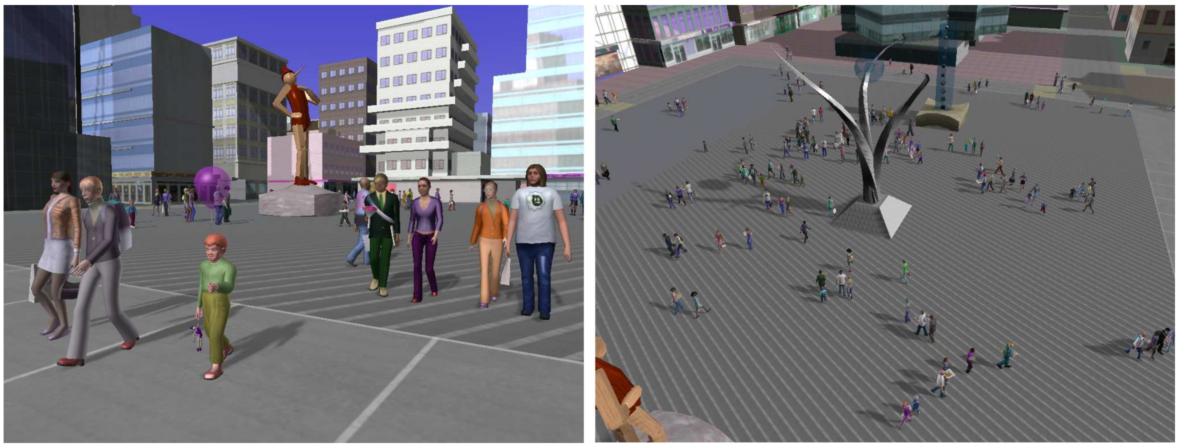


Figure 10.2: Pedestrians are widely distributed in the scene, thanks to the planner (see Figure 3.1) that finds a large variety of paths for a given set of original/destination points. Groups of 2 to 5 people are also created to further increase the realism of the simulation.

10.2 Implementation of Segmentation Maps

The color variety technique described in Chapter 4 can be mainly implemented in a fragment shader. We here detail how to do it in GLSL, using Shader Model 3.0 hardware. Note that for the sake of clarity and brevity, the following code shows how to apply color variety with an appearance set composed of a single segmentation map, *i.e.*, each fragment can at most belong to four different parts. However, once the concept is grasped, it is easily extended to multiple segmentation maps.

10.2.1 Color Contributions

To facilitate the fragment shader comprehension, in a first step, we begin by describing the basic code applying color variation to every fragment of one instance. Then, in a second phase, we will add the necessary lines to include customized per-part illumination effects (specularity and reflection).

The uniforms needed for the basic color variation algorithm are the following:

```
uniform sampler2D texture;
uniform sampler2D segmentationMap;
uniform sampler2D clut;
uniform vec2      clutCoord;
```

The `texture` and `segmentationMap` sampler2D handles represent the components of the appearance set used for the current object instance.

Each instance possesses one set of four colors, *i.e.*, one per part. The `clut` texture contains the color sets of all instances in the scene, ordered in a particular way. Firstly, to simplify the CLUT addressing, colors belonging to a single instance are always placed contiguously on the same row of the CLUT texture. Secondly, we store a set of four RGB colors in only three texels to save space: since the CLUT is an RGBA texture, it is possible to store the three first colors in the RGB channels of three texels, and use their vacant A channel to store the RGB components of the fourth color.

Finally, the uniform `clutCoord` contains the 2D coordinates that index the first color of the current instance. Note that since colors of a same set are always contiguous in a same row, we can efficiently iterate through the current set by only increasing the x coordinate of `clutCoord`.

The following code snippet shows how to color fragments:

```
vec3 colorAccum           = vec3( 0.0 );
vec3 clutFetchFourth      = vec3( 0.0 );
float whiteContribution   = 1.0;

 const vec4 textureFetch =
     texture2D( texture, gl_TexCoord[ 0 ].st );
 const vec4 segMapFetch =
     texture2D( segmentationMap, gl_TexCoord[ 0 ].st );

// Iterate through the RGB channels of the segmentation map.
for ( int i = 0; i < 3; ++i )

   const vec4 clutFetch =
       texture2D(
          clut,
          vec2( clutCoord.x + INV_CLUT_RES*i, clutCoord.y )
       );

   clutFetchFourth.r = clutFetch.a;

   colorAccum += segMapFetch.r * clutFetch.rgb;

   whiteContribution -= segMapFetch.r;

   // Segmentation map and color accum
   // bonus texel swizzling.
```

```
27      segMapFetch = segMapFetch.gbar;
28      clutFetchFourth = clutFetchFourth.gbr;
29
30
31   // Get color accum bonus.
32   whiteContribution -= segMapFetch.r;
33   colorAccum        += segMapFetch.r * clutFetchFourth.rgb;
34
35   gl_FragColor.rgb =
36      textureFetch * ( max( colorAccum, vec3( 1.0 ) )
37             + vec3( max( whiteContribution, 0.0 )  )
```

At lines 01 to 03 we declare three local variables. The variable `colorAccum` will accumulate the weighted contribution of each part's color.

At lines 05 and 07, the texture and segmentation map are both addressed by the same implicit built-in varying `gl_TexCoord[0].st`, since they share the same uv parameterization. The corresponding fetched colors are saved in `textureFetch` and `segMapFetch`, respectively.

The core of the algorithm is a loop through the R, G, and B values of the segmentation map fetch.

First of all, at line 13, the color of the current part is retrieved from the CLUT. As previously explained, only the x coordinate of the CLUT needs to be incremented. Note that `INV_CLUT_RES` represents the reciprocal of the CLUT resolution, used to normalize the computed coordinates between 0.0 and 1.0.

Then, at line 19, the `clutFetchFourth` variable is used to save the fourth color in the CLUT. Recall that its RGB components are stored in the A channel of the CLUT's three texel set. This color is thus only known at the end of the loop, once the three CLUT texels have been read.

At line 21, the color contribution of the part currently processed is added: the current segmentation map channel is used to weight the `clutFetch`color contribution. The fourth part's color contribution, saved in `clutFetchFourth`, is only added at line 33, once out of the loop. Note that in a shader, the intensity of a texture channel is no longer expressed between 0 and 255, but normalized between 0.0 and 1.0. Thus, the RGB components of the resulting `colorAccum`are also in this range.

To iterate over the segmentation map's channels, a swizzling of the `segMapFetch` variable is achieved at line 27. Similarly, to save the fourth color's components, the `colorFetchFourth` variable is also swizzled at line 28.

Sometimes, it happens that the sum of contributions for a texel does not reach precisely 1.0 (introduced in Section 4.3.1). This is the case when, in a segmentation map, the sum R+G+B+A does not equal 255 for a texel. It happens particularly often when designing smooth transitions between two parts. In the above code, care is taken to treat this particularity. If the sum exceeds 255 (1.0 in the shader), it is simply clamped (see line 36). If however, the sum does not reach 255, the problem is trickier: from a design point of view, it means that a certain percentage of the texel belongs to no part at all. From a technical point of view, let us take an example: imagine a transition texel in a segmentation map. Its R and G components both equal 100 (39%), the others equal 0. For simplicity, let us assume the corresponding part colors in the CLUT are pure red and green. From the shader code above, the final color in `colorAccum`is 39% of red and 39% of green which represents a dark yellow:

$$\frac{100}{255}red + \frac{100}{255}green \simeq 0.39 \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \end{pmatrix} + 0.39 \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \end{pmatrix} = \begin{pmatrix} 0.39 \\ 0.0 \\ 0.39 \end{pmatrix} \tag{10.1}$$

This seems to be the correct calculation. However, with this approach, when trying to achieve a gradient transitioning smoothly from red to green, the result provides shades of yellow that are too dark. To compensate for this, we add white to `colorAccum` to fill in the missing contribution. In our example, 22% of white is added, resulting in a `colorAccum` of (0.61, 0.22, 0.61). In the code, at line 03, the variable `whiteContribution` is set to 1.0 (or 100%), and at lines 23 and 32, the percentage of other parts are substracted from it. The final color is computed at line 35, where the white contribution is added to `colorAccum`.

10.2.2 Multiple Materials

Now that the basic mechanism has been illustrated, we show how to obtain per-part illumination effects. Here, we illustrate the case of specular and sphere map reflection parameters. Firstly, we need three more uniforms:

```
uniform vec2      specularParams[ 4 ];
uniform sampler2D sphereMap;
uniform float     reflectivityParams[ 4 ];
```

To achieve specularity, from Phong's equation, two values are required: intensity and exponent. Thus, `specularParams` is an array of four 2D vectors, one for each part of the segmentation map. For reflection effects, the sphere map (reflection texture) is sent in `sphereMap`, and the reflectivity parameter in `reflectivityParams`. Once more, there are four of them, one per part. Note that we only send one sphere map, thus all parts of this segmentation map potentially reflect the same environment. The following code snippet is similar to the previous one except for a few additional lines, emphasized with a star (*).

```
01   vec3 colorAccum             = vec3( 0.0 );
02   vec3 clutFetchFourth        = vec3( 0.0 );
03   float whiteContribution     = 1.0;
04*  vec3 reflectionContribution = vec3( 0.0 );
05*  float specularContribution;
06
07    const vec4 textureFetch =
08        texture2D( texture, gl_TexCoord[ 0 ].st );
09    const vec4 segMapFetch =
10        texture2D( segmentationMap, gl_TexCoord[ 0 ].st );
11*   const vec3 sphereMapFetch =
12*       texture2D( sphereMap, gl_TexCoord[ 1 ].st );
13
14   // Iterate through the RGB channels of the segmentation map.
15   for ( int i = 0; i < 3; ++i )
16
```

```
17      const vec4 clutFetch =
18          texture2D(
19             clut,
20             vec2( clutCoord.x + INV_CLUT_RES*i, clutCoord.y )
21          );
22
23      clutFetchFourth.r = clutFetch.a;
24
25      colorAccum += segMapFetch.r * clutFetch.rgb;
26
27*     specularContribution +=
28*         segMapFetch.r *
29*         specularParams[ i ].x *
30*         pow( r0DotE, specularParams[ i ].y );
31*     reflectionContribution += segMapFetch.r *
32*         reflectivity[ i ] * sphereMapFetch;
33
34      whiteContribution -= segMapFetch.r;
35
36      // Segmentation map and color accum
37      // bonus texel swizzling.
38      segMapFetch = segMapFetch.gbar;
39      colorAccumBonus = colorAccumBonus.gbr;
40
41
42   // Get color accum bonus.
43   whiteContribution -= segMapFetch.r;
44*  specularContribution   +=
45*      segMapFetch.r *
46*      specularParams[ 3 ].x *
47*      pow( rDotE, specularParams[ 3 ].y );
48*  reflectionContribution +=
49*      segMapFetch.r *
50*      reflectivity[ 3 ] *
51*      sphereMapFetch;
52   colorAccum += segMapFetch.r * clutFetchFourth.rgb;
53
54   gl_FragColor.rgb =
55      textureFetch * ( max( colorAccum, vec3( 1.0 ) )
56             + vec3( max( whiteContribution, 0.0 )  ) ;
57*  gl_FragColor.rgb +=
58*      reflectionContribution + vec3( specularContribution );
```

At line 09, the sphere map is addressed with the built-in varying `gl_TexCoord[1].st`, previously computed in the vertex shader (see vertex shader code in the Appendix). The fetched color is saved in `sphereMapFetch`.

Then, in the loop, at line 27, the variable `specularContribution` is computed as the weighted contribution of each part's specularity. The applied formula is the one commonly used in the Phong model.

Similarly for the reflection, at line 31, the variable `reflectionContribution` accumulates all the parts' contributions as their reflectivity multiplied by the fetched sphere map color, weighted by the channel intensity in the segmentation map. At lines 44 and 48, the fourth part's contribution to specularity and reflection is accumulated. Note that it would be impossible to integrate the fourth part's contribution in the loop, since we are constrained to three iterations corresponding to the CLUT's sets of three colors.

The presented code uses a loop, available only in Shader Model 3.0. However, by unrolling the loop, this technique can also be implemented on older GPUs. Note that it is exactly what a good compiler should do, even on Shader Model 3.0 parts.

LIST OF FIGURES

1.1 *YaQ* is a software architecture dedicated to the real-time simulation of large crowds of varied virtual humans. 9

1.2 Five human templates taking full advantage of accessories and segmentation maps. 12

2.1 *(left to right)* A deformable mesh, skinned and animated in real time; a rigid mesh, *i.e.*, pre-computed posture of the same mesh; and an impostor, a pre-computed image of the same mesh, textured onto a quad to give the illusion of a 3D mesh. 16

2.2 Results obtained in [Wand and Straßer, 2002], using a hierarchy of points and triangles. 17

2.3 Dobbyn et al. combined rigid meshes with impostors to render large crowds and keep a high appearance quality close to the camera [Dobbyn et al., 2005]. 18

2.4 A polypostor is a 2D polygonal representation of a virtual human. Its animation is only based on the displacement of its vertices [Kavan et al., 2008]. . 18

2.5 Variety in appearance is achieved by dividing a human into several body parts and coloring each part with varied colors. The body parts are differentiated by a specific intensity in the alpha channel of the texture [Tecchia et al., 2002a]. 20

2.6 Crowd using color variety and decals from [Gosselin et al., 2004]. 21

2.7 Results obtained in [Galvao et al., 2008] by varying the color, limb size, muscle bulge, and body mass of characters. 21

2.8 The physical simulation of clothes is pre-computed and re-used on rigid meshes and impostors [Dobbyn et al., 2006]. 22

2.9 Reynolds takes advantage of the parallel processors of the PS3 to simulate large crowds of fishes with an agent-based method [Reynolds, 2006]. 23

2.10 Outdoor navigation of pedestrians in [Lamarche and Donikian, 2004]. Path planning is based on the topological properties of the environment, while inter-pedestrian avoidance is ruled by a neighborhood graph. 24

2.11 In [Treuille et al., 2006], crowds are steered by a potential field to reach their goals while avoiding collisions. 24

2.12 ABS: the Agent Behavior Simulator, rules pedestrian behaviors based on several 2D maps of the environment [Tecchia et al., 2001]. 25

2.13 Group of characters socializing in [O'Sullivan et al., 2003]. 26

3.1 The architecture of YaQ is divided into three components. In a first step, the Variety component (in green) and the Navigation component (in red) precompute the necessary data (instances and waypoints). In a second step, these data are fed to the Real-time component (in blue). 29

3.2 By applying variety techniques at three levels, instances of a same template seem unique. Color variations are used to create clothes and skin patterns, shape variations are introduced with accessories, such as wigs or glasses, and various animations are exploited for further individualization. 30

3.3 Images of walking humans. *(Left)* The textured and shaded results, as rendered in *YaQ*. *(Center)* The RGB channels of the image are used to represent the 3D directions of the normals: Red in the X direction, Green in the Y direction, and Blue in the Z direction. *(Right)* The RG channels of the image are used to represent the texture coordinates. 33

3.4 A 1024×1024 atlas storing the UV map (above) and the normal map (below) of a virtual human performing a keyframe of an animation from several points of view. 34

3.5 Storage space in $[Mb]$ on a logarithm scale for one second of animation of each level of detail: *(a)* deformable mesh, *(b)* rigid mesh, *(c)* impostor. . . . 34

3.6 Shared resources between representations of a human template: *(a)* deformable mesh, *(b)* rigid mesh, *(c)* impostor. 36

4.1 Five different human templates: their shape, textures, skeletons are different. 40

4.2 Five different textures mapped onto five instances of the same human template. 41

4.3 We instantiate several times the same human template with the same texture. Only per-body-part color variety is used to modify their appearance. 42

4.4 Texture used for color variety. *(Left)* the RGB channels represent the basic texture, which is very bright to allow for color modulations. *(Right)* the alpha channel of the texture is used to differentiate body parts. Each body part takes a specific alpha intensity. 43

4.5 *(a)* Human instances have their body part colors randomly chosen. *(b)* For each body part, a designer has constrained the range of possible colors using our dedicated tool, based on the HSB color model. 44

4.6 HSB color space. Hue is represented by a circular region. A separate square region may be used to represent saturation and brightness, *i.e.*, the vertical axis of the square indicates brightness, while the horizontal axis corresponds to saturation. 44

4.7 The HSB space is constrained to a three dimensional color space with the following parameters *(a)* hue from 20 to 250, saturation from 30 to 80 and brightness from 40 to 100. *(b)* Colors are then randomly chosen inside this space to add variety on the eye texture of a character. 45

4.8 Close-up on the transition between skin and hair: artifacts in previous methods *(a)* with bilinear filtering and *(b)* nearest filtering. *(c)* Smooth transitions obtained with segmentation maps and bilinear filtering. *(d)* Detailed transition between skin (red channel) and hair (green channel) on the segmentation map. 46

4.9 *(top to bottom)* a human template original texture; several pairs of segmentation maps sharing the same parameterization; examples of unique sets of eight colors for each character; detailed effects obtained with segmentation maps and specularity parameters on faces (make-up, freckles, glossy lips, etc.) and on the whole body (cloth patterns, shiny shoes, etc.). 47

4.10 A blue to red gradient. *(Left)* The sum of the red and blue contributions does not reach 255 in some pixels, causing the gradient to suffer from an unwanted black contribution, *(Right)* A white contribution is added so that the sum of contributions is always 255. 48

4.11 A CLUT image used to store the color of each virtual human body parts and accessories (detailed in Section 4.3.2). 49

4.12 Appearance sets applied to instances of a single human template. Note the different specular effects on the body parts and the varying cloth patterns. . 50

5.1 Two human templates wearing the same hat, in their default posture. The pink, yellow and blue points represent the position and orientation of the root, the head joint $(m1)$, and the hat accessory $(m2)$, respectively. 56

5.2 *(a)* Sampled tiles for a hat impostor creation. *(b)* The resulting uv map and *(c)* normal map. 58

5.3 *(a)* Pipeline of accessory impostor positioning. *(b)* Impostors with no occlusion treatment. *(c)* Impostors with our fragment depth computation. 59

5.4 *(Left)* a human template in default posture. *(Right)* An instance of the human template playing an animation clip. The displacement of the body, relatively to the world origin $(m3)$ is depicted in red, the displacement of the head joint due to the animation clip $(m4)$ in yellow. 63

5.5 *(Left)* a representation of the accessory repository, sorted by type. Each accessory possesses its own list of body (or human instance) ids. Reciprocally, all bodies possess slots filled with their assigned accessories. *(Right)* an illustrated example of the accessory slots for body with id 1. 65

5.6 Several instances of a single human template, varied through the appearance sets and several segmentation maps, and accessories. 65
5.7 The space of interactive height and shape variety at creation. The human template grows up in the right axis, while it gets more fat on the left axis. . 67
5.8 Two FatMaps designed in 3DSMax [Autodesk, 2009a]. Dark areas represent regions more influenced by fat or muscle modification, while lighter parts are less modified. 68
5.9 Results of shape and height modifications shown on multiple instances of a single template. 70

6.1 Example of motion kit structure. On the left, a virtual human instantiated from a human template points to the motion kit it currently uses. In the center, a motion kit with its links identifying the corresponding animations to use for all human templates and LOD. 75
6.2 A virtual human using a motion kit with identifiers: *[walk] [none] [cell-phone] [pocket]*. 76
6.3 Examples of upper body movements (hands in the pocket, phone call, hand on hip, ...) added to a generic locomotion cycle. 78
6.4 Set of controlled effectors attached to the hand and corresponding goal positions attached to the pelvis. 79
6.5 Example of posture from an accessorized locomotion cycle. 80
6.6 Results obtained in an urban environment when modifying the upper body at runtime to make virtual humans carry bags, puppets, balloons, flowers, etc. 82
6.7 More complex accessories carried by virtual humans in a theme park environment. 82

7.1 The real-time component of *YaQ* architecture is composed of four steps: the Scaler, the Simulator, the Animator, and the Renderer. 84
7.2 Dense crowd in a large environment. 89
7.3 Shadowed scene with apparent directional light frustum. 90
7.4 Frames per second obtained for (*top*) highly detailed deformable meshes, and (*bottom*) simple deformable meshes. The red lines show the results obtained when working with sorted lists, the green ones with a naive approach. The stars indicate the results for 30 frames per second. 93
7.5 Frames per second obtained for (*top*) rigid meshes, and (*bottom*) impostors. The red lines show the results obtained when working with sorted lists, the green ones with a naive approach. The stars indicate the results for 30 frames per second. 94
7.6 (*left*) Virtual humans navigating in a complex environment. (*right*) Same configuration of virtual humans with apparent levels of detail; in red: the rigid meshes, in green: the impostors. 95

8.1 Relations between data structures. Each mesh instance possesses a set of colors in the CLUT. Applying colors smoothly to the texture parts is achieved with the use of segmentation maps. 98

8.2 Each instance of a building template randomly chooses an appearance set (a texture and a segmentation map). 98

8.3 Appearance sets applied to instances of six building template. 99

8.4 Appearance sets composed of two segmentation maps (eight body parts) applied to instances of six human templates. Accessories are varied with appearance sets composed of one segmentation map (maximum four different parts). 100

8.5 Unique characters simulated in a theme park. 101

8.6 Number of displayable characters at 30 fps with 0 to 3 accessories for *(a)* deformable meshes, *(b)* rigid meshes, *(c)* impostors, and *(d)* impostors with depth computation. 102

8.7 Virtual Romans simulated in a reconstructed district of Ancient Pompeii. The colors worn by instances reflect their status: slaves, poor, rich, etc. . . . 103

8.8 (*left*) A middle-class Virtual Roman carries amphoras in her hands and also on her head, while (*right*) a rich Virtual Roman may carry bread in her hands, but would not bear anything on her head. 104

8.9 Virtual Romans walking in a street of Ancient Pompeii. 104

8.10 Jérôme Grapinet, physical therapist, has created a new exercise for his patients, using *YaQ*. 105

8.11 *(left)* Mr. Grapinet's projection cabin (in pink) has no right angles, and allows a patient to feel fully immersed in the projected virtual world. *(right)* The projection cabin used to display the crowd simulation of *YaQ*. 106

8.12 The CAVE of the EPFL-VRLAB uses simple home video projectors to immerse a user in a virtual environment. Head tracking enables tracking a user's orientation and position inside the CAVE allowing for interaction with the walking virtual humans. 107

9.1 *(top) YaQ* simulating and navigating a crowd of varied virtual humans in a city environment. *(bottom)* The 3 different rendering representations become apparent: deformable meshes (in red), rigid meshes (in green) and impostors (in blue). 113

9.2 Appearance sets applied to instances of a single human template. Note the different specular effects on the body parts and the varying cloth patterns. . 114

9.3 Virtual humans can carry complex accessories, such as handbags and balloons. This effect is achieved by freezing and clamping specific joints of the upper body at runtime. 114

10.1 A navigation graph is composed of circles representing navigable areas. When two circles intersect, pedestrians are allowed to navigate from one navigable area to the other. 116

10.2 Pedestrians are widely distributed in the scene, thanks to the planner (see Figure 3.1) that finds a large variety of paths for a given set of original/destination points. Groups of 2 to 5 people are also created to further increase the realism of the simulation. 117

BIBLIOGRAPHY

Adobe. http://www.adobe.com/products/photoshop/photoshop/, 2009. 4.3.1

B. Allen, B. Curless, and Z. Popović. Exploring the space of human body shapes: Data-driven synthesis under anthropometric control. *SAE transactions*, 113(1):245–248, 2004. 2.2.2

A. Aubel, R. Boulic, and D. Thalmann. Real-time display of virtual humans: Levels of detail and impostors. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(2): 207–217, 2000. 2.1.2, 2.1.2

Autodesk. http://usa.autodesk.com/adsk/servlet/index?siteid=123112&id=5659302, 2009a. 5.5.1, 5.8, 10.2.2

Autodesk. http://usa.autodesk.com/adsk/servlet/index?id=6837710&siteid=123112, 2009b. 6.3

P. Baerlocher and R. Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *Vis. Comput.*, 20(6):402–417, 2004. 1.1.3, 6.3

J. Barczak, N. Tatarchuk, and C. Oat. GPU-based scene management for rendering large crowds. In *ACM SIGGRAPH Asia Sketches*, 2008. 2.1.2

G. Le Bon. *Psychologie des Foules*. 1895. 2

S. Chenney. Flow tiles. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 233–242, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. 2.3

P. De Heras Ciechomski, B. Ulicny, R. Cetre, and D. Thalmann. A case study of a virtual audience in a reconstruction of an ancient Roman odeon in Aphrodisias. In *The 5th*

International Symposium on Virtual Reality, Archaeology and Cultural Heritage, 2004. 2.1

R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, 1986. 7.4.1

P. de Heras Ciechomski, S. Schertenleib, J. Maïm, D. Maupu, and D. Thalmann. Real-time Shader Rendering for Crowds in Virtual Heritage. In *VAST '05, 2005*, 2005. 4.1

S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry / impostor crowd rendering system. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 95–102, New York, NY, USA, 2005. ACM Press. 2.1.2, 2.3, 2.2.1, 3.2.4, 5.2, 10.2.2

S. Dobbyn, R. McDonnell, L. Kavan, S. Collins, and C. O'Sullivan. Clothing the masses: Real-time clothed crowds with variation. In Charles Hansen and D Fellner, editors, *Proceedings of Eurographics Short Papers*, pages 103–106. Eurographics Association, 2006. 2.2.2, 2.8, 10.2.2

B. Dudash. Skinned instancing. In *NVidia Direct3D SDK 10 Code Samples*, 2007. 2.2.2

A. Egges, T. Di Giacomo, and N. Magnenat-Thalmann. Synthesis of realistic idle motion for interactive characters. In *Game Programming Gems 6*, 2006. 3.2.2

R. Galvao, R. G. Laycock, and A. M. Day. Gpu techniques for creating visually diverse crowds in real-time. In *VRST '08: Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, pages 79–86, New York, NY, USA, 2008. ACM. 2.2.1, 2.2.2, 2.7, 10.2.2

T. Giang, R. Mooney, C. Peters, and C. O'Sullivan. Aloha: Adaptive level of detail for human animation: Towards a new framework, eurographics 2000 short paper proceedings, pp 71Ð77. *Eurographics 2000, Short Presentations*, 2, 2000. 2.4

P. Glardon, R. Boulic, and D. Thalmann. PCA-based walking engine using motion capture data. In *Proc. of Computer Graphics International*, 2004a. 1.1.3, 3.1, 3.3.2, 6.1.2, 9.4

P. Glardon, R. Boulic, and D. Thalmann. A coherent locomotion engine extrapolating beyond experimental data. In *Proc. of Computer Animation and Social Agent*, 2004b. 1.1.3, 3.1, 3.3.2, 6.1.2, 9.4

M. Gleicher, H. J. Shin, L. Kovar, and A. Jepsen. Snap-together motion: assembling run-time animations. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–9, New York, NY, USA, 2008. ACM. 2.4

D. Gosselin, P. V. Sander, and J. L. Mitchell. Drawing a crowd. In Wolfgang Engel, editor, *ShaderX3: Advanced Rendering Techniques in DirectX and OpenGL*. Charles River Media, Cambridge, MA, 2004. 2.2.1, 2.6, 10.2.2

J. Grapinet. `http://www.vestibulaire.net/Realite-virtuelle_a30.html`, 2008. 8.4.1

N. Greene. Environment mapping and other applications of world projections. *IEEE Comput. Graph. Appl.*, 6(11):21–29, 1986. 5.2.2, 5.2.2

J. Hamill, R. McDonnell, S. Dobbyn, and C. O'Sullivan. Perceptual evaluation of impostor representations for virtual humans and buildings. *Computer Graphics Forum (Eurographics '05)*, 24(3), 2005. 2.1.2

L. Heigeas, A. Luciani, J. Thollot, and N. Castagné. A physically-based particle model of emergent crowd behaviors. In *Graphicon*, 2003. 2.3

D. Helbing, P. Molnár, and F. Schweitzer. Computer simulations of pedestrian dynamics and trail formation. *Evolution of Natural Structures*, pages 229–234, 1994. 2.3

D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, September 2000. 2.3

R. L. Hughes. The flow of human crowds. *Annual Review of Fluid Mechanics*, 35(1):169–182, 2003. 2.3

R.L. Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36:507–535(29), 2002. 2.3

M. Kasap and N. Magnenat-Thalmann. Parameterized human body model for real-time applications. In *CW '07: Proceedings of the 2007 International Conference on Cyberworlds*, pages 160–167, Washington, DC, USA, 2007. IEEE Computer Society. 2.2.2

Ladislav Kavan, Simon Dobbyn, Steven Collins, Jiri Zara, and Carol O'Sullivan. Polypostors: 2d polygonal impostors for 3d crowds. In *2008 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 149–155. ACM Press, February 2008. 2.1.2, 2.4, 5.2.2, 6.4, 9.3, 10.2.2

A. Kirchner and A. Shadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. In *Physica A*, pages 237–244, 2001. 2.3

F. Lamarche and S. Donikian. Crowd of virtual humans: a new approach for real time navigation in complex and structured environments. *Computer Graphics Forum*, 23(3): 509–518, 2004. 2.3, 2.10, 10.2.2

M. Lau and J. J. Kuffner. Precomputed search trees: Planning for interactive goal-driven animation. In *2006 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 299–308, September 2006. 2.3

K. Hoon Lee, M. Geol Choi, Q. Hong, and J. Lee. Group behavior from video: a data-driven approach to crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 109–118, 2007. 2.3

W. S. Lee and N. Magnenat-Thalmann. Virtual body morphing. In *The Fourteenth Conference on Computer Animation*, pages 158–166, 2001. 2.2.2

A. Lerner, Y. Chrysanthou, and D. Lischinski. Crowds by example. *Eurographics'07: Computer Graphics Forum*, 26(3), 2007. 2.3

M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985. 2.1.1

D. Luebke, B. Watson, J. D. Cohen, M. Reddy, and A. Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002. 3.2

N. Magnenat-Thalmann, R. Laperreire, and D. Thalmann. Joint dependent local deformations for hand animation and object grasping. In *Graphics Interface*, pages 26–33, 1988. 3.2.2

J. Maïm, S. Haegler, B. Yersin, P. Müller, D. Thalmann, and L. Van Gool. Populating Ancient Pompeii with Crowds of Virtual Romans. In *The 8th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST'07)*, pages 109–116, 2007. 8.3

J. Maïm, B. Yersin, M. Clavien, and D. Thalmann. `http://www.youtube.com/watch?v=ZA3gzqG94JQ`, 2007. 8.3

J. Maïm, B. Yersin, M. Clavien, and D. Thalmann. `http://www.youtube.com/watch?v=puJAuQp6aUM`, 2009. 8.2.1

J. Maïm, B. Yersin, and D. Thalmann. To appear: Unique instances for crowds. *IEEE Computer Graphics and Applications*, 2009. 2.1.2, 2.2.1

R. McDonnell, S. Dobbyn, and C. O'Sullivan. LOD human representations: A comparative study. In *Proceedings of The First International Workshop on Crowd Simulation (V-CROWDS'05)*, 2005. 2.1.2

R. McDonnell, S. Dobbyn, S. Collins, and C. O'Sullivan. Perceptual evaluation of lod clothing for virtual humans. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 117–126. Eurographics Association, 2006. 2.2.2

R. McDonnell, M. Larkin, S. Dobbyn, S. Collins, and C. O'Sullivan. Clone attack! perception of crowd variety. *ACM Trans. Graph.*, 27(3):1–8, 2008. 2.2.1, 6

E. Millan and I. Rudomin. Impostors and pseudo-instancing for gpu crowd rendering. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 49–55, New York, NY, USA, 2006. ACM. ISBN 1-59593-564-9. 2.1.2, 3.2.4

T. Möller and E. Haines. *Real-time rendering*. A. K. Peters, Ltd., Natick, MA, USA, 1999. 7.4.1

P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 614–623, 2006. 8.3

S. R. Musse and D. Thalmann. A hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):152–164, 2001. 2.4

S. R. Musse, C. Babski, T. Capin, and D. Thalmann. Crowd modelling in collaborative virtual environments. In *Proc. ACM VRST*, pages 115–123, 1998. 2.4

nvidia. Nvidia geforce 8800 gpu architecture overview, 2006. 3.2.2

C. O'Sullivan, J. Cassell, H. Vilhjámsson, J. Dingliana, S. Dobbyn, B. McNamee, C. Peters, and T. Giang. Levels of detail for crowds and groups. *Computer Graphics Forum*, 21(4): 733–741, 2003. 2.4, 2.13, 10.2.2

S. Paris, J. Pettré, and S. Donikian. Pedestrian steering for crowd simulation: A predictive approach. In *Eurographics'07*, 2007. 2.3

N. Pelechano and N. I. Badler. Modeling crowd and trained leader behavior during building evacuation. *IEEE Comput. Graph. Appl.*, 26(6):80–86, 2006. 2.4

N. Pelechano, J.M. Allbeck, and N.I. Badler. Controlling individual agents in high-density crowd simulation. In *SCA'07*, 2007. 2.4

J. Pettré, P. de Heras Ciechomski, J. Maïm, B. Yersin, J.-P. Laumond, and D. Thalmann. Real-time navigating crowds: scalable simulation and rendering. *Comput. Animat. Virtual Worlds*, 17(3-4):445–455, 2006. 2.3, 3.1, 7.1.2, 10.1

J. Pettré, H. Grillon, and D. Thalmann. Crowds of moving objects: Navigation planning and simulation. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2007. 2.3, 3.1, 10.1

W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1987. ACM Press. 7.4.1

C. W. Reynolds. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM Press. 2.3, 2.9, 10.2.2

C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM Press. 2.3

C. W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, 1999. 2.3, 7.2

I. Rudomin and E. Millan. Point based rendering and displaced subdivision for interactive animation of crowds of clothed characters. In *VRIPHYS 2004: Virtual Reality Interaction and Physical Simulation Workshop*, pages 139–148, 2004. 2.1.1

G. Ryder and A. M. Day. Survey of real-time rendering techniques for crowds. *Comput. Graph. Forum*, 24(2):203–215, 2005. 2.1, 2.2.2

G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques'97*, pages 151–162, London, UK, 1997. Springer-Verlag. 2.1.2

H. Seo, L. Yahia-Cherif, T. Goto, and N. Magnenat-Thalmann. Genesis: Generation of e-population based on statistical information. In *CA '02: Proceedings of the Computer Animation*, page 81, Washington, DC, USA, 2002. IEEE Computer Society. 2.2.2

H. Seo, F. Cordier, and N. Magnenat-Thalmann. Synthesizing animatable body models with parameterized shape modifications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 120–125, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 2.2.2

W. Shao and D. Terzopoulos. Autonomous pedestrians. In *Proc. ACM SIGGRAPH/EG SCA*, pages 19–28, 2005. 2.4

A. R. Smith. Color gamut transform pairs. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 12–19, New York, NY, USA, 1978. ACM Press. 4.2.2

M. Sung, M. Gleicher, and S. Chenney. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3):519–528, 2004. 2.4

M. Sung, L. Kovar, and M. Gleicher. Fast and accurate goal-directed motion synthesis for crowds. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 291–300, New York, NY, USA, 2005. ACM Press. 2.3

F. Tecchia and Y. Chrysanthou. Real-time rendering of densely populated urban environments. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 83–88, London, UK, 2000. Springer-Verlag. 2.1.2, 2.4

F. Tecchia, C. Loscos, R. Conroy, and Y. Chrysanthou. Agent behavior simulator (abs): A platform for urban behaviour development. In *GTEC'2001*, pages 17–21, 2001. 2.4, 2.12, 10.2.2

F. Tecchia, C. Loscos, and Y. Chrysanthou. Image-based crowd rendering. *IEEE Computer Graphics and Applications*, 22(2):36–43, 2002a. 2.1.2, 2.2.1, 2.5, 3.2.4, 10.2.2

F. Tecchia, C. Loscos, and Y. Chrysanthou. Visualizing crowds in real-time. *Computer Graphics Forum*, 21(4):753–765, December 2002b. 2.1.2, 2.2.1

A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, New York, NY, USA, 2006. ACM. 2.3, 2.11, 7.2, 10.1.2, 10.2.2

B. Ulicny, P. de Heras Ciechomski, and D. Thalmann. Crowdbrush: interactive authoring of real-time crowd scenes. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 243–252, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. 1.1.2

Y. Uralsky. Efficient soft-edged shadows using pixel shader branching. In *GPU Gems 2*, 2005. 7.4.1

M. Wand and W. Straßer. Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum*, 21(3), 2002. 2.1.1, 2.2, 10.2.2

L. Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM Press. 7.4.1

A. Woo, P. Poulin, and A. Fournier. A survey of shadow algorithms. *IEEE Comput. Graph. Appl.*, 10(6):13–32, 1990. ISSN 0272-1716. 7.4.1

A. Yershova and S. M. LaValle. Deterministic sampling methods for spheres and SO(3). In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3974–3980, 2004. 5.2.2

B. Yersin. *Real-Time Motion Planning, Navigation, and Behavior for Large Crowds of Virtual Humans*. PhD thesis, EPFL, 2009. 1.2, 3.1, 10.1

B. Yersin, Jonathan Maïm, F. Morini, and D. Thalmann. Real-time crowd motion planning: Scalable avoidance and group behavior. *Vis. Comput.*, 24(10):859–870, 2008. 10.1.2

Jonathan MAIM

43, Av. du Chablais
1008 Prilly, Switzerland
Tel: +41 78 652 62 64

Email: jonathan.maim@gmail.com
Web: http://vrlab.epfl.ch/~maim/
Linkedin: http://www.linkedin.com/in/jonmaim

Education

May 2005 – May 2009	**Swiss Federal Institute of Technology** (EPFL) – Lausanne, Switzerland PhD student at the Virtual Reality Laboratory (VRLAB).
Aug. 2004 – Mar. 2005	**University of Montréal** (UdeM) – Montréal, Canada Visiting student at Computer Graphics Group (LIGUM).
1999 – 2005	**Swiss Federal Institute of Technology** (EPFL) **–** Lausanne**,** Switzerland Masters Degree in Computer Science.

Work experience

May 2005 – May 2009	**Swiss Federal Institute of Technology** (EPFL) – Lausanne, Switzerland Research assistant

Skills

Programming:	C++, GPU (GLSL, Cg), Java, Python, Lua, SQL, PHP, HTML, XML.
Platforms:	Linux and Windows.

Extracurricular courses

Feb. - Jun. 2008	**Venture challenge Start-Up Creation Course**, Unil Lausanne, Switzerland.
Sep. 2007 – Jun. 2008	**Accounting and Finance Course**, EPFL, Lausanne, Switzerland.
Sep. – Dec. 2007	**Entrepreneurial Opportunity Identification and Exploitation**, EPFL, Lausanne, Switzerland.
Feb. – Jun. 2004	**CREATE Entrepreneurship Course**, EPFL, Lausanne, Switzerland.

Languages

French:	Mother tongue.
English:	Fluent.
German:	School knowledge.

Publications

MAIM, J., YERSIN, B., THALMANN, D. ***Unique Instances for Crowds.*** Accepted with minor revisions to IEEE Computer Graphics and Applications Journal, 2008.

MAIM, J., YERSIN, B., PETTRE, J., THALMANN, D. ***YaQ: An Architecture for Real-Time Navigation and Rendering of Varied Crowds.*** IEEE Computer Graphics & Applications Special Issue July/August on Virtual Populace, 2009.

MAIM, J., YERSIN, B., PETTRE, J., THALMANN, D. ***Crowd Patches: Populating Large-Scale Virtual Environments for Real-Time Applications.*** ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, February 27-March 1, 2009.

MAIM, J., YERSIN, B., THALMANN, D. ***Improved Color Variety for Geometry Instancing using Segmentation Maps.*** Chapter in ShaderX6, Wolfgang Engel (Eds.), Charles River Media, February 12, 2008.
http://www.amazon.com/Shader-X6-Rendering-Wolfgang-Engel/dp/1584505443/

MAIM, J., HAEGLER, S., YERSIN, B., MUELLER, P., THALMANN, D., VAN GOOL, L. ***Populating Ancient Pompeii with Crowds of Virtual Romans.*** The 8th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST'07), Eurographics Proceedings, Brighton, UK, November 26-30, 2007.

MAIM, J., YERSIN, B, DE HERAS CIECHOMSKI, P. ***An architecture for Real-Time Crowds of Virtual Humans.*** In Crowd Simulation, Thalmann D. and Musse S. R. (Eds.), Springer, October 30, 2007.
http://www.amazon.com/Crowd-Simulation-Daniel-Thalmann/dp/184628824X

YERSIN, B., MAIM, J., DE HERAS CIECHOMSKI, P. ***Crowd Appearance Variety.*** In Crowd Simulation, Thalmann D. and Musse S. R. (Eds.), Springer, October 30, 2007.
http://www.amazon.com/Crowd-Simulation-Daniel-Thalmann/dp/184628824X

MORINI, F., YERSIN, B., MAIM, J., THALMANN, D. ***Real-Time Scalable Motion Planning For Crowds.*** In proceedings of the 2007 International Conference on Cyberworlds, IEEE Computer Society, Hannover, Germany, 24-26 October, 24-26 October, 2007.

PETTRE, J., DE HERAS CIECHOMSKI, P., MAIM, J., YERSIN, B., SCHERTENLEIB, S., THALMANN, D., LAUMONT, J.-P. ***Real-Time Navigating Crowds: Scalable Simulation and Rendering.*** In Computer Animation and Virtual Worlds (CAVW) Journal, 17(3-4):445-455, CASA 2006 Special Issue, 2006.

YERSIN, B., MAIM, J., DE HERAS CIECHOMSKI, P., SCHERTENLEIB, S., THALMANN, D. ***Steering a Virtual Crowd Based on A Semantically Augmented Navigation Graph.*** In First International Workshop on Crowds Simulation (V-CROWDS'05), Lausanne, Switzerland, November 24-25, 2005.

DE HERAS CIECHOMSKI, P., SCHERTENLEIB, S., MAIM, J., MAUPU, D., THALMANN, D. ***Real-Time Shader Rendering for Crowds in Virtual Heritage.*** In the 6th International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST'05), Pisa, Italy, November 8-11, 2005.

DE HERAS CIECHOMSKI, P., SCHERTENLEIB, S., MAIM, J., THALMANN, D. ***Reviving the Roman Odeon of Aphrodisias: Dynamic Animation and Variety Control of Crowds in Virtual Heritage.*** In 11th International Conference on Virtual Systems and Multimedia (VSMM'05), Ghent, Belgium, October 3-7, 2005.

MAIM, J. ***Radiosity-based Illumination from Major 3D Polygonal Contributors in Immersive Walkthroughs.*** Master Thesis, 2005.