HAL

archives-ouvertes.fr

Omnidirectional texturing of human actors from multiple view video sequences

Alexandrina Orzan

▶ To cite this version:

Alexandrina Orzan. Omnidirectional texturing of human actors from multiple view video sequences. Graphics [cs.GR]. 2005. inria-00598504

HAL Id: inria-00598504
https://hal.inria.fr/inria-00598504

Submitted on 6 Jun 2011

Omnidirectional texturing of human actors from multiple view video sequences

Alexandrina ORZAN

Master internship report

Artis - GRAVIR/IMAG-INRIA UMR CNRS C5527.

Jury composition :

Jean-Marc　HASENFRATZ　Directeur de stage

Contents

Résumé

Ces dernières années, de plus en plus d'activités de recherche sont consacrées à l'étude de la vidéo tridimensionnelle, créée à partir de plusieurs flux vidéo. Le but est d'obtenir une vidéo *free-viewpoint*, où l'utilisateur peut observer d'un point de vue arbitraire, choisi de manière interactive, une scène filmée par plusieurs caméras.

Les applications possibles sont diverses. Un système *free-viewpoint* peut augmenter le réalisme visuel de la technologie de téléprésence[1]. De ce fait des utilisateurs situés physiquement en différents endroits peuvent collaborer à travers un même environnement virtuel. En outre, les effets spéciaux employés par l'industrie du film, comme ceux introduits dans le film Matrix (*freeze-and-rotate*), seraient rendus accessibles à tous les utilisateurs.

Dans la plupart des applications de réalité virtuelle, nous cherchons à représenter des acteurs sous la forme d'avatar. C'est pourquoi la recherche est importante dans ce domaine.

Pour les vidéos de type *free-viewpoint*, la scène est filmée simultanément par différentes caméras depuis plusieurs points de vue. Les flux vidéo obtenus par les caméras sont utilisés pour créer un modèle 3D de la scène. Cette reconstruction tridimensionnelle est indispensable pour que l'utilisateur puisse regarder la scène depuis n'importe quel point de vue. Dans le cadre de la réalité virtuelle, il est possible d'ajouter de nouveaux objets dans cette scène (objets virtuels) et de traiter les problèmes d'éclairage (ombres au sol, . . .), ainsi que les problèmes d'occultation [7, 8].

Le modèle 3D peut être décrit en utilisant différentes méthodes, telles que des maillages, des échantillons de points ou des voxels. Pour rendre le modèle plus réaliste, les flux vidéo provenant des caméras sont plaqués sur le modèle 3D. Finalement, en combinant le modèle 3D reconstruit et les différents flux vidéo, nous sommes capables de reconstruire un monde virtuel réaliste.

Le but du stage effectué a été de réaliser le "texturage" en temps réel d'un modèle

---

[1]"Telepresence technology" enables people to feel as if they are actually present in a different place or time (S. Fisher & B. Laurel, 1991) or enables objects from a different place to feel as if they are actually present (T. Lacey & W. Chapin, 1994).

3D d'un animateur. L'étude a été effectuée dans le cadre du projet CYBER-II[2]. Ce projet vise à simuler, en temps réel (au minimum 25 images par secondes), la présence d'une personne (par exemple un présentateur de télévision ou un professeur) dans un environnement virtuel.

Concernant la reconstruction 3D, nous distinguons généralement deux approches : la reconstruction *model-free*, où l'on ne fait pas d'hypothèse à priori sur la forme de l'objet et la reconstruction *model-based* pour laquelle nous connaissons à l'avance les caractéristiques de l'objet.

La reconstruction *model-free*, ne faisant aucune supposition sur la géométrie de la scène, nous pouvons traiter des scènes dynamiques et complexes. Dans le cas d'une personne, cette méthode permet de tenir compte, par exemple, des mouvements des vêtements ou des cheveux. La plupart de ces méthodes *model-free* cherchent à construire le contour des objets [13]. Pour cela, les silhouettes de l'objet sont extraites dans chaque image en détectant les pixels se trouvant au premier plan. A partir de ces différents contours, nous pouvons reconstruire une représentation 3D de l'objet. Nous parlons alors d'*enveloppe visuelle*. Cette enveloppe peut être représentée sous la forme de voxels [8] ou sous la forme d'un ensemble de polygones [14, 17, 16].

L'approche *model-free* permet une reconstruction et un rendu temps réel. Cependant, elle nécessite l'utilisation d'un grand nombre de caméras pour obtenir une reconstruction précise et fidèle.

La reconstruction *model-based* suppose que l'on connaisse les caractéristiques de l'objet observé. Dans notre cas nous supposons que c'est une personne et nous cherchons à faire correspondre les différentes silhouettes à une représentation générique d'un corps humain [2, 9, 10]. Cette approche a l'avantage de produire un modèle 3D précis, mais est peu flexible. En effet, elle ne peut prendre en compte les mouvements des habits, ou d'autres objets non prévus à priori.

Dans le projet CYBER-II, nous utilisons une méthode *model-free* pour la reconstruction, avec six flux vidéo et un modèle polyédrique de la scène.

Pour augmenter le réalisme, il est indispensable d'habiller le modèle. Pour cela, les flux vidéo sont plaqués sur la géométrie 3D [3]. Les méthodes pour effectuer ce "multi-texturage" proposées dans la littérature sont classées en deux grandes catégories : *view-dependant* et *view-independant*.

L'habillage *view-dependant* est fondé sur l'image qui sera affichée à l'écran. Cette image est construite à partir des caméras les plus proches de l'observateur. Quand celui-ci est situé entre différentes caméras, deux à quatre textures sont mélangées afin d'obtenir l'image finale [4, 5]. Cette méthode introduit un certain nombre d'erreurs visibles, en particulier dans les parties o la géométrie du modèle ne correspond pas exactement à la forme observée. De plus, les résultats sont souvent flous et le changement de caméra dû au déplacement de l'observateur peut être visible.

---

[2]http ://artis.imag.fr/Projects/Cyber-II/

Concernant l'approche *view-independant*, nous utilisons l'ensemble des caméras pour habiller la totalité du modèle. Dans ce cas, nous ne tenons pas compte de la position de l'observateur. Dans cette méthode nous choisissons la caméra la plus appropriée pour chaque polygone du modèle [2, 9, 16]. L'avantage de cette méthode réside dans le fait qu'une texture est associée une fois pour toute à un polygone. Il n'y a donc plus de discontinuité dans l'affichage d'un même polygone lors du changement de point de vue de l'observateur. D'autre part, le flou dû à un mélange de texture n'existe plus. Cependant, des discontinuités entre polygones adjacents peuvent être visibles. Ce cas de figure apparaît lorsque des caméras différentes sont utilisées pour des polygones d'une même région. Pour atténuer cette discontinuité, nous pouvons mélanger (*blending*) les textures provenant des différentes caméras. Les paramètre utilisés lors du mélange peuvent être calculés par sommet ou par polygone [2, 4, 5]. Matsuyama [16] propose de déterminer la couleur pour chaque sommet, puis de "recolorier" les triangles avec les couleurs obtenues en interpolant linéairement les valeurs RGB des sommets. Cependant, pour de grands triangles, les petits détails comme des plis dans les vêtements sont perdus. Li et de Magnor [15] calculent le mélange pour chaque pixel de l'image finale, ce qui permet un mélange plus précis.

Pour obtenir un texturage correct, il est nécessaire de déterminer quels points sont visibles depuis chaque caméra. Si une caméra ne voit pas une certaine région, alors elle ne devrait pas être employée dans le calcul de la couleur pour cette région. Afin de résoudre le problème de visibilité, Debevec [4] divise les triangles du modèle de sorte qu'ils soient entièrement visibles ou entièrement invisibles depuis toutes les positions des caméras. Matusik [17] propose de déterminer la visibilité de chaque sommet lors du calcul de l'enveloppe visuelle sans aucun coût supplémentaire. Cependant des erreurs existent lorsque tous les sommets d'un polygone sont visibles alors que celui-ci est partiellement recouvert. Magnor et al. [15] résolvent le problème de visibilité par fragment, en utilisant une technique de *shadow mapping*. Cependant, ils ont besoin d'effectuer deux rendus de la scène pour chaque caméra. De plus, l'approche n'est pas temps réel même avec une implémentation utilisant massivement les capacités des cartes graphiques.

A partir des méthodes d'habillage décrites ci-dessus, nous avons proposé trois algorithmes de "multi-texturage". L'objectif était de trouver le meilleur compromis qualité/temps.

Comme nous disposons d'un ensemble de caméras, plusieurs vues du même point sont disponibles. De ce fait et à cause des imperfections du modèle, plusieurs problèmes sont à résoudre :

– comment obtenir des transitions douces entre les triangles adjacents, de sorte que les bords ne soient pas visibles ?
– comment choisir l'ensemble de caméras qui contribueront à la texture ?
– comment décider quelle partie du modèle 3D est visible pour chaque caméra ?

– comment assurer la cohérence temporelle, de sorte que les couleurs demeurent les mêmes d'une image à l'autre pour chaque partie du modèle ?

Nous nous sommes concentrés sur des algorithmes décidant la couleur de chaque pixel indépendamment les uns des autres, choix justifié par les contraintes imposées par le modèle.

Le premier algorithme implémenté a été un algorithme *view-dependant*. Dans cette première approche nous avons cherché à éviter le traitement de la visibilité. Pour cela, nous avons comparé, pour un même point, les couleurs provenant de chaque caméra et nous avons conservé uniquement la couleur la plus souvent présente. Par exemple, si pour un point sur la surface, trois caméras voient du rose et une quatrième voit du gris, nous pouvons considérer que la couleur correcte est le rose. Nous avons montré que comparer les couleurs est une chose pertinente dans notre cas. Nous nous sommes basés sur l'écart type afin d'éliminer les couleurs erronées. Cette approche permet ainsi un traitement intrinsèque de la visibilité.

La deuxième approche sur laquelle nous avons travaillé utilise un algorithme *view-independant*. Le but était de résoudre le problème de discontinuité entre les polygones adjacents.

La solution proposée élimine une grande partie des fausses couleurs. Cependant, des régions visibles par moins de trois caméras ne sont pas considérées dans le calcul de d'écart type, et des erreurs se produisent.

Une autre méthode de "multi-texturage" sur laquelle nous avons travaillé emploie un algorithme *view-independant* pour choisir les meilleurs caméras pour chaque triangle de la surface du modèle. Un test de visibilité est fait pour chacun de ces triangles, pour décider s'ils font face à la caméra ou non. Si un triangle est *back-facing*, il est invisible du point de vue de la caméra ; mais s'il est *front-facing*, il pourrait être visible ou caché par un autre triangle. Dans ces cas, nous employons l'écart type pour trancher.

Les méthodes *view-independant* "fixent" les texture une fois pour toute pour chaque polygone. Pour les régions où l'objet est vu par au moins trois caméras, l'algorithme réussit à éliminer les fausses couleurs et a mélanger les couleurs restantes sans que l'on ne s'aperçoivent que les textures proviennent de différentes caméras. Cependant, pour les régions où l'objet est vu par au plus deux caméras, l'algorithme fonctionne sans élimination de couleur.

Pour améliorer ces résultats, nous avons appliqué un filtre médian à l'image finale. Cette variation de l'algorithme lisse la texture et empêche le changement rapide de couleur dans le temps, mais elle double le temps de calcul.

Une dernière méthode emploie la technique de *shadow-mapping* pour déterminer, pour chaque caméra, les points cachés par d'autres objets. Elle utilise le mode *view-independant* de texturage pour combiner l'information de couleur provenant des diverses caméras qui voient ce point. L'utilisation de la couleur dominante

permet de supprimer divers artefacts. Pour de petites parties non visibles, le filtre médian est employé pour compléter la couleur.

Dans le contexte du projet CYBER-II, nous avons observé que pour habiller le modèle nous n'utilisons généralement que cinq caméras. En effet, la sixième caméra est généralement située à l'opposé du point de vue de l'observateur. Nous avons donc développé une technique rapide choisissant les cinq caméras les plus pertinentes qui seront les seules utilisées.

En conclusion, nous avons proposé d'éliminer les fausses couleurs en faisant un test d'écart type. Les résultats sont plutôt bons grâce à la correction des artefacts de visibilité et à la suppression des erreurs induites par l'utilisation d'un modèle imparfait.

Nous avons utilisé une version modifiée du filtre médian pour remplir les petites régions invisibles. La méthode proposée fonctionne dans le plan image et n'a pas besoin de l'information géométrique.

La méthode *view-independant* que nous avons utilisée, couplée au choix de la couleur dominante, réussit à éliminer les fausses couleurs pour les pixels vus par plus de deux caméras, sans faire une vérification coûteuse d'occultation. Cependant, le système à six caméras que nous avons employé n'offre pas l'information suffisante pour que cet algorithme fonctionne sans erreur. Il existe des cas où seulement deux caméras, ou moins, voient un point.

L'algorithme qui emploie la technique de *shadow-mapping* est mieux adapté à notre système. Combiné avec l'écart type et la méthode pour remplir les "trous", il produit des textures très réalistes. C'est un algorithme qui convient à un nombre restreint de caméras. Puisqu'il dépend du nombre de lumières permis par OpenGL, il est limité à un maximum de huit caméras. Pour plus de caméras, l'algorithme *view-independant* pourrait probablement habiller correctement un objet sans avoir recours à la vérification des occultations avec une technique de *shadow-mapping*.

Ce mémoire est organisé en six chapitres. Après une introduction du domaine de la vidéo 3D, une présentation détaillée des problèmes de recherche rencontrés pendant ce stage est faite (chapitre 2). Un état de l'art est proposé dans le chapitre 3. Le chapitre 4 est consacré aux méthodes proposées de "multi-texturage". Le chapitre 5 est une discussion des résultats et le chapitre 6 conclut ces recherches et annonce les travaux futurs.

Chapter 1

Introduction

Currently, visual media such as television and motion pictures only present a two-dimensional impression of the real world. In the last few years, increasingly more research activity has been devoted to investigate three-dimensional video, created from multiple camera views. The goal is to have a free-viewpoint video, where the user is able to watch a scene from an arbitrary viewpoint chosen interactively.

The possible applications are manifold. A free-viewpoint system can increase the visual realism of telepresence technology [1], thus enabling users from different locations to collaborate in a shared, simulated environment as if they were in the same physical room. Also, special effects used by the movie industry, such as *freeze-and-rotate* camera, would be made accessible to all users.

For free-viewpoint video, a scene is typically captured from several viewpoint by different video cameras working simultaneously. From the views obtained by the cameras a 3D model is created. The re-creation of the real objects is necessary, because the purpose is not simply to anticipate how the scene would look from an arbitrary point of view, but to integrate the real objects into a virtual scene, with virtual obstacles and lights. In order to do so, we have to be able to compute the way the model is lighted by the virtual lights and how it shadows the virtual objects, and this is impossible without knowing the 3D geometry [7, 8] (See Figure 1.1 for an example of 3D model integrated into a virtual world).

The 3D shape can be described using various methods, such as polygon meshes, point samples or voxels. To make the model more realistic, images captured from the video streams are typically mapped onto the 3D shape, thus completing the virtual representation of the real object.

Since people are central to most visual media content, research has been dedicated in particular to the extraction and reconstruction of human actors.

---

[1]"Telepresence technology" enables people to feel as if they are actually present in a different place or time (S. Fisher & B. Laurel, 1991) or enables objects from a different place to feel as if they are actually present (T. Lacey & W. Chapin, 1994).
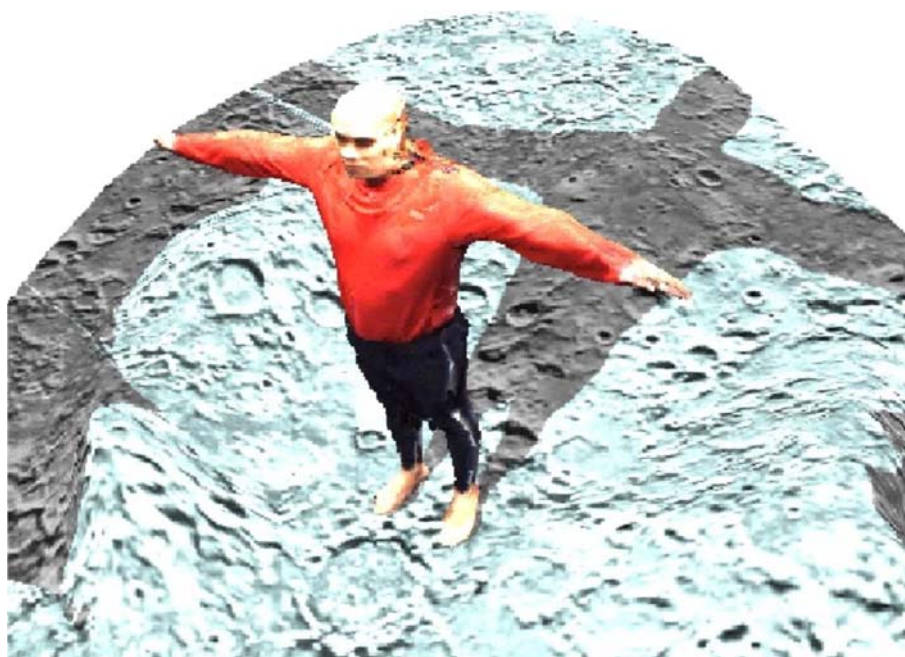
Figure 1.1: Recovered 3D model of a human actor integrated in a virtual environment with virtual lighting.

The rest of the paper proceeds with a detailed presentation of the research problems encountered during this internship, in chapter 2. A review of related work is done in chapter 3, while chapter 4 will be dedicated to describing the proposed methods of texture-mapping. The remaining chapters discuss the results and future tasks. Part of this internship's work was described in an article at the Romanian Conference on Computer-Human Interaction, article that we included as an appendix.

Chapter 2

Problem statement

During this internship, the goal was to texture in real-time the recreated 3D model of a moving actor. The study was conducted within the context of CYBER-II project[1], which aims to simulate, in real-time (at least 25 images/second), the presence of a person (e.g. a TV presenter or a teacher) in a virtual environment.

The system used by CYBER-II has 6 cameras, positioned as seen in Figure 2.1.

The person can move freely in a cube of approximatively $8m^3$, where it is captured on film by the video cameras; the geometrical form is then reconstructed in real time. The acquisition is not restricted to a single human actor. Moreover, the system allows the reconstruction of multiple persons and objects present in the scene.

The computed geometrical form has to be textured with the images recovered from the cameras. As we dispose of a set of cameras, we have more then one view for the same patch of the 3D model. In addition, the geometry of the model is limited to about 5000 triangles, not sufficient to have a good mesh representation, but necessary if we want to obtain it in real time. This small number of triangles makes the texturing errors more visible.

Several problems arise:

- how to obtain smooth transitions between adjacent patches, so that the edges are not visible.

- how to choose the set of cameras that will contribute to the texture.

- how to decide which part of the 3D model is visible from each camera.

- how to ensure temporal coherence, so that the colors remain the same from one frame to another for each part of the model.

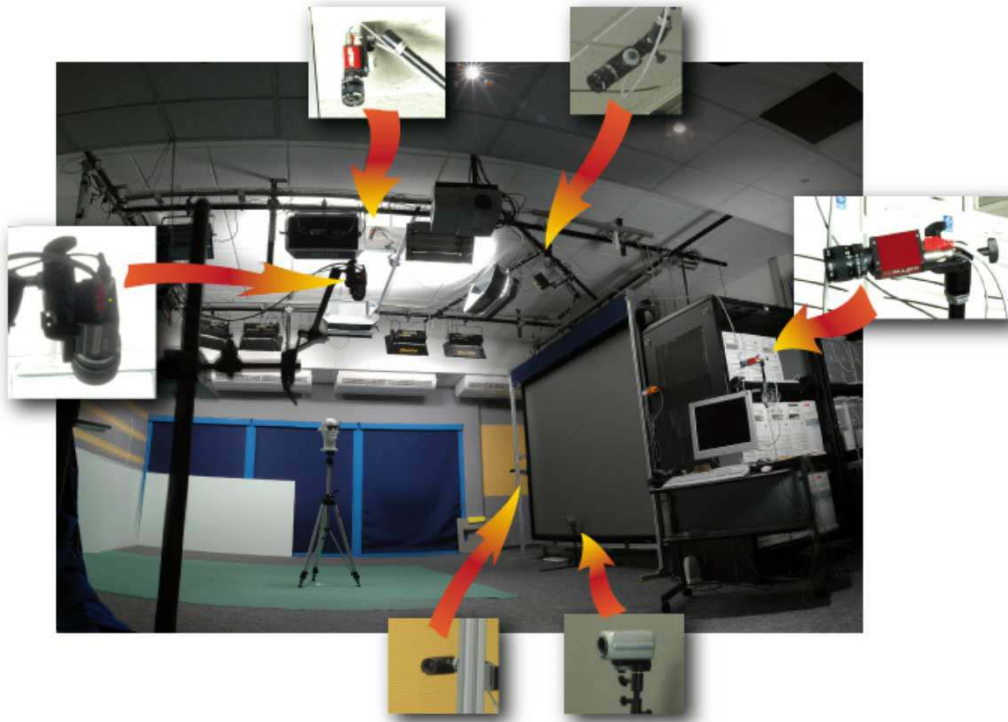These were the problems that we tried to solve during this internship.

---

[1]http://artis.imag.fr/Projects/Cyber-II/

Figure 2.1: Camera setting

2.1   Texture junction

Given that we have 6 views of the object, the textures on neighboring patches are often extracted from different images. This introduces jitters at patch boundaries, as seen in Figure 2.2.

Ensuring continuity at the patches frontier is a very important problem, since without it the final result has important errors.

2.2   Choice of the cameras

Usually, for a specific point of view, not all cameras need to be considered. There are some cases when a camera sees little or nothing of the currently visible part of the model. These cameras need to be identified, especially if we are dealing with a greater number of cameras, and considering all of them would make impossible reaching real-time.

The problem here is to decide quickly how much of the visible image a camera sees, and if it's not the only camera that sees a certain part of the model. Moreover,

Figure 2.2: Two examples of the texture junction problem. Note how visible the texture borders are.

when we zoom out the object, some parts of the model become almost invisible, and some cameras could be disregarded.

Figure 2.3 shows a case when a camera view brings almost no contribution to the current view.

2.3 Visibility

Deciding what part of the 3D model is visible to each reference view is a very important problem for multi-view texture mapping. For those parts that are invisible in a reference view, the information should be ignored when doing the texturing. Otherwise, important errors will be obtained in the final result. Such an error can be seen in Figure 2.4, where the image of a hand is textured on the body.

2.4 Temporal coherence

The geometry of the model is decided for each frame in part and there is no connection between models derived from successive frames. Therefore, we cannot color each patch at the beginning, and then just follow how the patch moves in time. We have to decide for each frame, separately, what color the final pixel will have. This gives place to fast color-variations. More precisely, parts of the 3D model may change color for each image, which is very disturbing for the viewer.
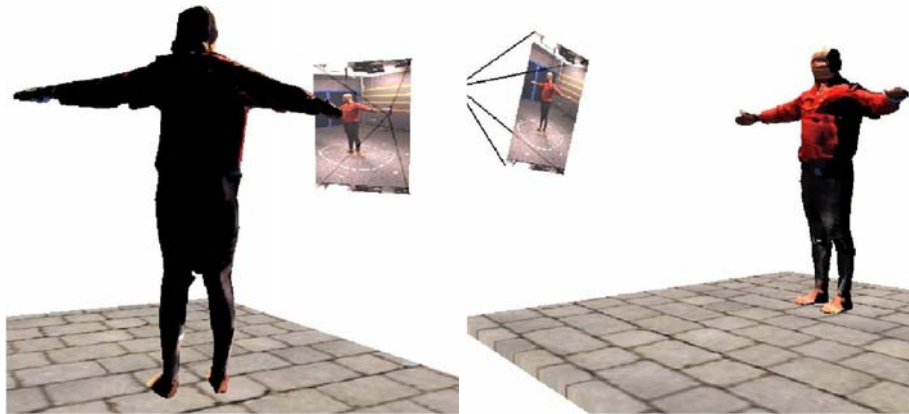
(a) A camera that doesn't bring much contribution to the current point of view.

(b) The same camera is used, but the scene is viewed from a different position.

Figure 2.3: Snapshots illustrating how a camera contribution varies for different views.
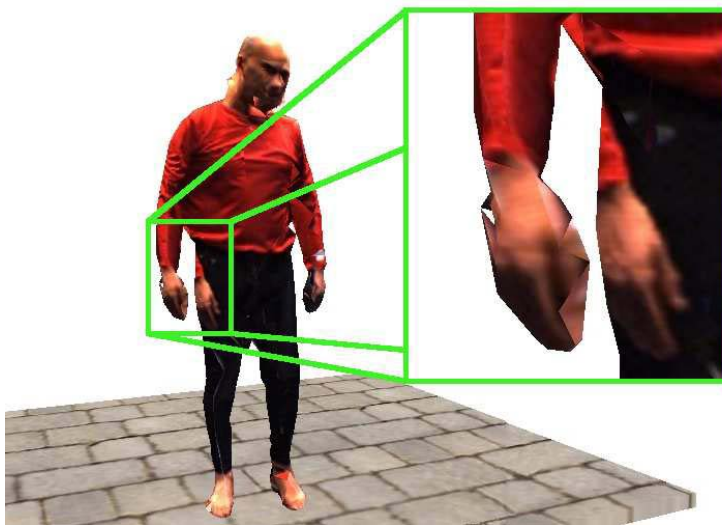
Figure 2.4: Visibility problem example. See the two right hands textured on the model.

An example of two consecutive frames, with different colors for the same zone, can be observed in Figure 2.5.

Consequently, we need to develop a method for preserving the color in a way con-

(a) Texturing of the first frame

(b) Texturing of the second frame

(c) Zoom in of the first frame texturing

(d) Zoom in of the second frame texturing

Figure 2.5: Temporal coherence problem: two consecutive frames with different texture color for the same region.

sistent with the model movements, without considering geometrical temporal coherence.

Chapter 3

State of the Art in Omnidirectional Texturing

Three-dimensional production from multiple view video, or 3D video, was first popularized by Kanade et. al. [11, 18], who proposed to obtained an immersive visual medium that lets the viewer select his viewing position. Kanade named this medium *Virtualized Reality*, since it *virtualizes* the event in order to permit the free movement of the user.

3.1 3D Model reconstruction

Two different approaches of model reconstruction have been studied in the recent years: model-free and model-based reconstruction.

Model-free reconstruction makes no a priori assumptions on scene geometry, allowing the reconstruction of complex dynamic scenes. In human modeling it allows the reproduction of detailed dynamics for hair and loose clothing.

Most model-free methods aim to estimate the visual hull, an approximate shell that envelopes the true geometry of the object [13]. To achieve this, object silhouettes are extracted from each camera image by detecting the pixels not belonging to the background.

The visual hull can then be reconstructed either by voxel-based or polyhedron-based approaches. The first approach discretizes a confined 3D space in voxels and carves away those voxels whose projection fall outside the silhouette of any reference view [8]. Polyhedron-based approaches represent each visual cone as a polyhedral object and computes the intersection of all visual cones [14, 17, 16]. Examples of different model-free object reconstructions are given in Figure 3.1.

The visual hull allows real-time reconstruction and rendering, yet it needs a large
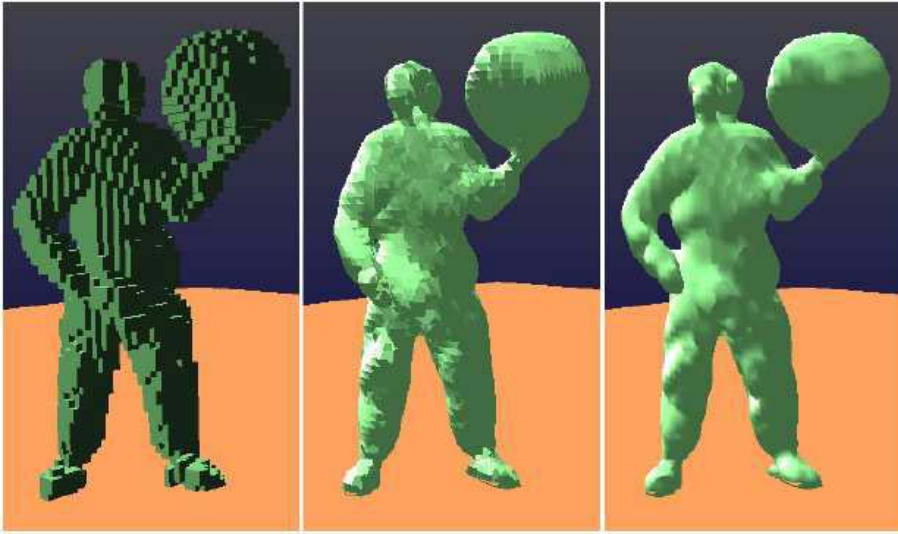
Figure 3.1: Model-free methods. From left to right: cube model, pure marching cubes surfaces, smoothed surfaces (image taken from [8]).

number of views to accurately represent a scene, otherwise the obtained model is not very exact.

Model-based reconstruction assumes that the real object is a human body and uses a generic humanoid model, which is deformed to fit the observed silhouettes [2, 9, 10]. Although it results in a more accurate model and permits motion tracking over time, this approach is restricted to a simple model and does not allow complex clothing movements. Moreover, it places a severe limitation on what can be captured (i.e. a single human body) and it is not real-time.

3.2 Multi-view texture mapping

Original images from multiple viewpoint are often mapped onto recovered 3D geometry in order to achieve realistic rendering results [3]. Proposed methods for multi-texture mapping are either view-dependent or view-independent.

View-dependent texture mapping is based on the idea that what the viewer sees is the "real view", and it is best represented by the camera views closest to him. Therefore, it takes into consideration only the cameras nearest the current viewpoint. In between camera views, two to four textures are blended together in order to obtain the current view image [4, 5]. This method exhibits noticeable blending artifacts in parts where the model geometry does not exactly correspond to the observed shape. What's more, the result is usually blurred and the passing from one camera view to another does not always go unnoticed.

View-independent texture mapping main idea is that the "real view" of a flat surface is that seen from a point on the normal in the center of the surface. Consequently, it selects the most appropriate camera for each triangle of the 3D model, independently of the viewer's viewpoint [2, 9, 16]. The advantage of this method is that it does not change the triangle texture when the user changes the viewpoint. Moreover, the blurred effect is less noticeable. However, the problem is that the best camera is not the same from patch to patch, even if they are neighboring. Here also, blending between visible views is necessary in order to reduce the abrupt change in texture at triangle edges.

Blending is done using various formulas that depend of:

1. the angle α between the surface normal and the vector towards the considered camera

2. the angle β between the surface normal and the vector towards the viewpoint

3. the angle γ between the vector towards a camera and the vector towards the viewpoint
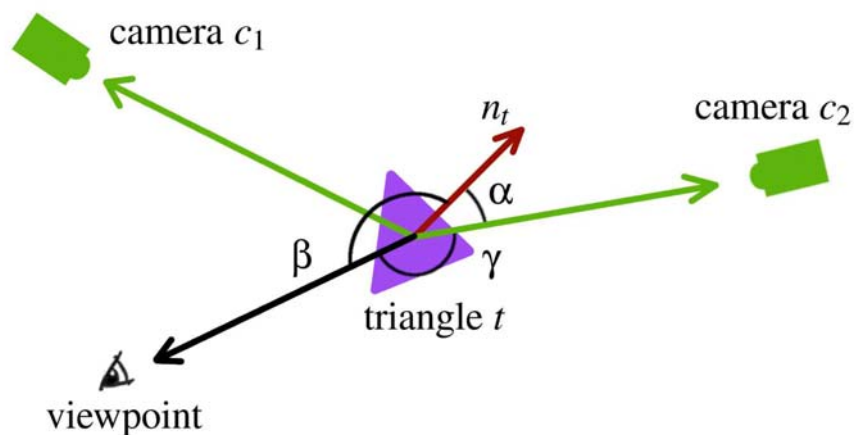
Figure 3.2: Angles considered while computing blending weights.

Blending weights can be computed per vertex or per polygon [2, 4, 5]. Matsuyama [16] proposes using this method for determining each vertex color and then paints the triangles with the colors obtained by linearly interpolating the RGB values of its vertices. However, for large triangles, small details like creases in the clothes are lost.

Li and Magnor [15] compute the blending for each rasterized fragment, which results in a more accurate blending.
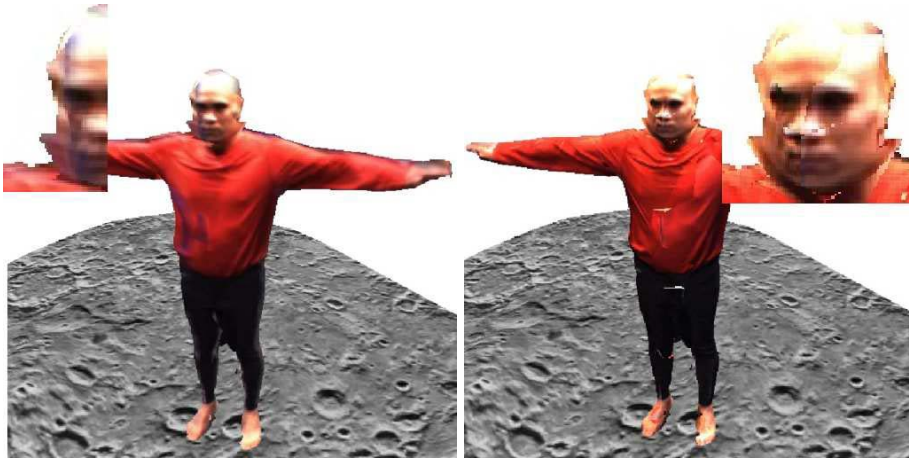
(a) A view-dependent texturing. Note the bluish color on the left (color taken from the background)

(b) The same model with a view-independent texturing. See the rapid change of color on the face and the right part of the body

Figure 3.3: View-dependent vs. view-independent texturing

3.3   Visibility

As we mentioned before, deciding visibilities with respect to camera views is a necessary task, because there are often cases where the camera's view of a surface point is occluded by another part of the geometric model. In these cases, the camera should not be used in computing the point's color.

In order to solve the visibility problem, Debevec et al. [4] splits the object triangles so that they are either fully visible or fully invisible to any source view. This process takes a long time even for a moderately complex object and is not suitable for real-time applications.

Matusik [17] proposes computing the vertex visibility at the same time that the visual hull is generated, at virtually no additional cost. Still, this does not guarantee that if the vertices are visible, the whole triangle is. Magnor et al. [15] solves the visibility problem per fragment, using shadow mapping. However, they require rendering the scene twice from each input camera viewpoint and is not real-time even with a hardware-accelerated implementation.

For the invisible triangles, Debevec [4] proposes assigning to the vertices the color of the closest visible triangle and using the Gouraud interpolation technique to fill in the triangle. Matsuyama [16] uses the linear interpolation to fill in the triangle

with the vertices color, if at least one of the vertices is visible; if not, the triangle remains black.

3.4   Temporal coherence

Temporal coherence refers to keeping track of how each object geometry varies in time and coloring the 3D model accordingly.

Vedula and Kanade [21] compute the scene flow, a measure of the scene motion, is a three-dimensional vector filed defined for every voxel. This scene flow describes how a voxel moves over time. The idea is that any 2D flow observed by a camera is the projection of the corresponding 3D flow of the point, so, if we have two or more cameras viewing a particular point, the scene flow can be recovered from the 2D images. Once a voxel evolution in time is known, the color can then be "controlled" so it doesn't have "unnatural" variations. The 3D model used is a voxel-based one. However, computing the scene flow is very slow (1.5 minutes for a single pair of frames).

Theobalt and Magnor [20] use a predefined human model that is overlapped on the computed silhouettes. Then the scene flow is reconstructed by searching through various possible motions of the model's joints. This approach restrains the search space considerably, but it also limits the scene to a single human. It still takes about 50 seconds to compute the scene flow between two consecutive frames.

Chapter 4

Texture Mapping Methods

In this chapter we present the different methods of texture-mapping we implemented. We start with a description of the 3D geometrical model used, briefly explaining how it was built and what constrains it imposes on our texture mapping algorithms. Next, we specify the OpenGL method employed for determining the texture coordinates in the object space. Finally, we discuss every texture-mapping method separately, giving an overview of the algorithm and commenting on the results.

4.1 The 3D model and the constraints it imposes

The 3D model employed during this internship is one created in the context of CYBER-II project and it's a polyhedron-based model obtained in real-time.

The method used to reconstruct the real scene is a model-free one that doesn't restrict the scene to a single human actor, like the model-based methods. However, since the purpose of the project was the reconstruction of a person, we did our research on a human model.

In order to recreate the geometry of the real object, our system computes the 2D silhouettes of the object, as viewed by each camera, and uses these to estimate the 3D shape.

Silhouette extraction is done by first acquiring the static background, as it is seen by every camera. Once the background is known, the pixels whose value differs from the corresponding background color are detected and treated as belonging to the silhouette. The result is a series of black and white pictures representing the silhouettes seen by each camera (as in Figure 4.1).

For every extracted silhouette, we specify the silhouette contour as a set of edges joining consecutive vertices. Then, for each camera view, the silhouette cone is
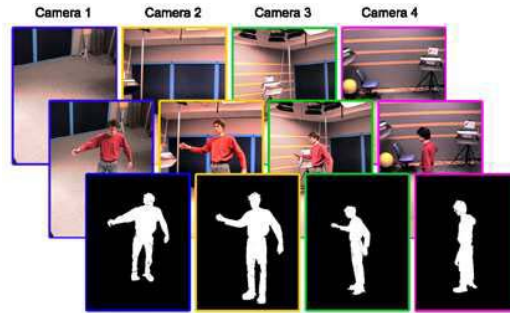
Figure 4.1: Silhouette extraction (image taken from [8])

calculated as the volume that originates from the camera's center of projection and extends infinitely while passing through the silhouette's contour (see Figure 4.2).

The *three-dimensional intersection* of all the silhouette cones will result in a set of polygons (in our case triangles) that define the surface of the visual hull. This surface, specified by the computed set of polygons, is the final 3D model (see Figure 4.2).
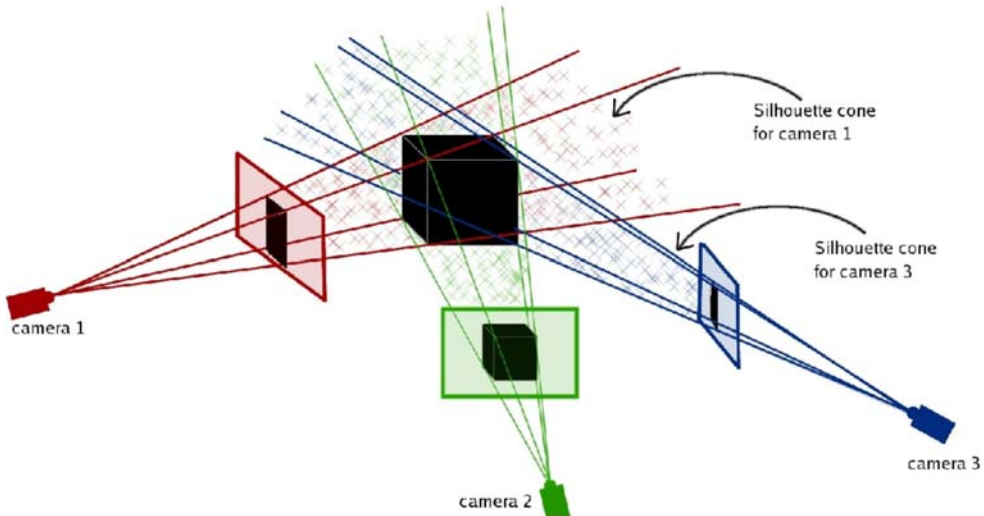
Figure 4.2: Computing the visual hull

The described method of creating the 3D object, while working in real-time, imposes some constraints on the resulting model. Firstly, it recreates the geometrical object at each frame. Secondly, the number of triangles, their form and position in space vary greatly in time, so we cannot track vertices from one frame to another.

This means that it is impossible to decide the color of the triangles only once, at the beginning of the video. Color values have to be computed in real-time, for each frame.

Moreover, the triangles which describe the model are not guaranteed to have a good aspect ratio (close to equilateral). There are cases when they are very elongated, as seen in Figure 4.3.
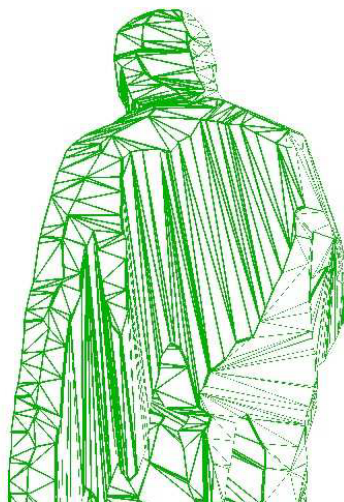
Figure 4.3: Example of elongated triangles, that are difficult to texture.

This poses problems in determining the visibility per triangle, since there are often cases where the triangle vertices are visible, but a part of the triangle is occluded. In order to correctly decide what parts are visible and what parts are not, we should check for visibility in every point, and not only at the triangle vertices and center.

Besides, considering each pixel separately would allow us to compute weighting information for each pixel, and not for a whole triangle or image.

This is why we decided to concentrate on per-pixel methods of texturing, rather than per-vertex or per-triangle methods.

4.2 Projective texture mapping that passes through geometry

To achieve realistic rendering results, we use the projective texture mapping, a method introduced by Segal [19] and included in the OpenGL graphics standard. Using this technique, the texture coordinates at a vertex are computed as the result of a projection rather than being assigned fixed values.

This method allows us to re-project the photograph of the real object back onto the geometry of the object. Still, the current hardware implementation of projective texture mapping in OpenGL lets the texture pass through the geometry and be mapped onto all back-facing and occluded polygons

Thus it is necessary to perform visibility check so that only regions visible to a particular camera are texture mapped with the corresponding image.

4.3 View-dependent method

The starting point for this method was the idea that we could avoid visibility checking by comparing the corresponding colors from the image views and eliminating those outside the general "trend". For instance, if at a point on the surface three cameras see rose and a fourth sees yellow, we could consider that the correct color is rose.

Next, we decided to try a view-dependent method, in the effort of having smooth continuous results by using the same image for adjacent triangles.

4.3.1 Eliminating the wrong colors

In order to be able to do multi-texturing, the supposition must be made that we are dealing with almost Lambertian surfaces. This means that the surfaces have mostly matte properties, their luminance being the same regardless of the viewing angle. Reflective materials, like brass or silver, are not Lambertian surfaces. All the same, this is not a very restrictive assumption, since most objects are not shiny. In the case of human reconstruction, for example, the skin and clothes are nearly Lambertian.

Furthermore, the cameras are calibrated prior to use, and the images are acquired at the same time and in the same lighting conditions.

All this makes it safe for us to suppose that if the cameras see the same point, they see it as having approximatively the same color. Therefore, we can compare colors and calculate distances in the RGB space; it is not necessary for us to switch to Lab color space, where we would be able to recognize the same object under different illumination [1, 6].

In order to decide which are the wrong colors, we use the standard deviation test. This test offers a measure of the degree of dispersion of the data from the mean value and allows us to eliminate the colors that are farther than expected from the average.

The test first computes the standard deviation σ of a set of n values $\mu_1 \ldots \mu_n$ as:

$$\sigma = \sqrt{\frac{1}{n} \cdot \sum_{k=1}^{n} (\mu_k - \mu)^2} \qquad \text{, where } \mu \text{ is the mean.}$$

The standard deviation is the expected variation around the average value, and is used in defining a confidence interval for which the values are considered plausible. Usually, these values are within one standard deviation away from the mean.

We use this test to eliminate the wrong colors, because even if an outlier moves the mean away from the main body of data, the standard deviation will indicate where the majority of values is situated. Still, in order to get a result, it is necessary that at least three cameras see the point and the majority sees the correct color.

If a sufficient number of color values are available for a pixel, we compute the mean (μ) and the standard deviation (σ) for each of the R, G, B channels. Individual colors falling outside the range $\mu \pm \beta \cdot \sigma$ for at least one channel are excluded. The factor β permits us to modify the confidence interval for which the colors are accepted. While the classical normal deviation test considers β is 1, we experimentally concluded that it was best to set it at 1.17, to allow for slight errors in manipulating the color-values.

If less than three possible colors are available for a pixel, we do not exclude any of them. If the pixel is invisible for all cameras, we compute its color using the color values of the neighbors whose color was already decided.

4.3.2   Choice of cameras

We select the cameras that observe the scene under an angle close to the current one. To do this, we consider the center of the scene to be the center of the model's bounding volume (the bright green point in Figure 4.4) and we compute the angle with which each camera deviates from the current viewpoint.
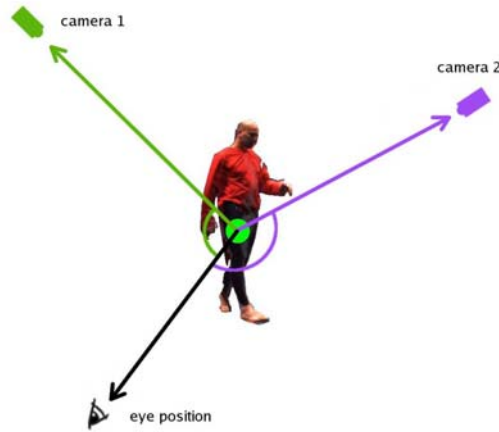
Figure 4.4: Angles between cameras and current viewpoint.

We define the penalty $penalty(c_i, v)$ as being the angular deviation of camera c_i from the current view v. We then do the interpolation by using the first k cameras having the smallest penalty. In our case, k was considered to be first 3, then 4.

To decide on the weight given to each camera, we have to define a weight function. This should give a maximum value to a camera with the penalty close to zero and

should drop to zero when the camera leaves the set of closest k. We therefore define the weight function as:

$$weight(c_i, v) = 1 - \frac{penalty(c_i, v)}{threshold(v)} \qquad \text{where } threshold(v) \text{ is the } penalty(c_{k+1}, v).$$

The threshold is considered to be the penalty of the first camera out of the chosen set in order to smooth the transition from one set of cameras to another. Normally, a camera leaves the set of the first k when the $(k+1)th$ camera becomes better, and including the latter into the already computing mean would make less visible the switch.

Since in OpenGL the maximum value for blending is 1, we normalize the weights to sum the unity, and the final weight function is:

$$\overline{weight}(c_i, v) = \frac{weight(c_i, v)}{\sum_{j=1}^{k} weight(c_j, v)}$$

So, once we decided which cameras to use, we can eliminate the wrong colors by using the algorithm described in the previous subsection. Each color that passes the test is then blended with the other contributing colors by using the corresponding camera weight. Thus, the final color is obtained.

4.3.3   Algorithm

The algorithm runs as described in Algorithm 4.3.1.

This method succeeds in eliminating a great deal of the wrong colors, as it can be seen in Figure 4.5. However, it isn't enough. There are cases where big portions of the model are invisible for all but one camera, and we texture them wrongly. There are also portions where only two cameras are near, and in this case we do not eliminate any of the colors.

Moreover, the passing from one camera set to another is still visible, even if it is less disturbing then for a naive implementation of view-dependent algorithm. This is because the nearest cameras are too far apart and we cannot smoothly interpolate between them.

We conclude that, although the implemented method might have satisfactory results for a larger number of cameras, for our system a visibility test is necessary. In addition, there is the problem of noticeable changes of camera views when rotating around the scene. For this, a view-independent method might be better.

4.4   View-independent method

This method of multi-texturing uses a view-independent algorithm to chose the best cameras for every triangle on the model's surface. A visibility test is perform

Input: A triangulated 3D mesh in the form of successive XYZ triplets describing the object surface.
Color images and camera parameters for each of the 6 cameras.
Output: A textured 3D model.

1 Find out the k nearest cameras;
2 **for** *each pixel in the image view* **do**
3   **if** *there are three or more cameras* **then**
4   Compute the mean and standard deviation;
5   **for** *each individual color* **do**
6   If it is not in the allowed interval, exclude;
7   **end**
8   **if** *there are colors left* **then**
9   Compute the weighted mean;
10   **end**
11   **else**
12   Compute the color using neighboring colors
13   **end**
14   **end**
15   **if** *there are less than three cameras* **then**
16   Compute the weighted mean;
17   **end**
18 **end**
19 Draw;

Algorithm 4.3.1: View dependent algorithm with selection of the dominant color

for each of these triangles, to decide whether they are facing the camera or not. If a triangle is back-facing, then it is invisible from the camera's point of view; but if it is front-facing, it could be visible or occluded by another triangle. For these cases, we use the method of eliminating wrong colors described in the previous section.

4.4.1 Visibility

A point p on the object's surface is visible from a camera c_i if:

1. the triangle t_j to which the point belongs faces the camera and

2. the point is not occluded by any other triangles.

The first condition can be fast determined by checking the truth of the inequality $n_{t_j} \cdot v_{c_i \to t_j} < 0$, where n_{t_j} is the triangle normal vector and $v_{c_i \to t_j}$ is the viewing direction from c_i towards the centroid of t_j (see Figure 4.6).

(a) Naive implementation of the view-dependent method.

(b) The view-dependent method after choosing the dominant color. Note how the blue color disappears

Figure 4.5: View dependent method before and after the wrong color elimination.
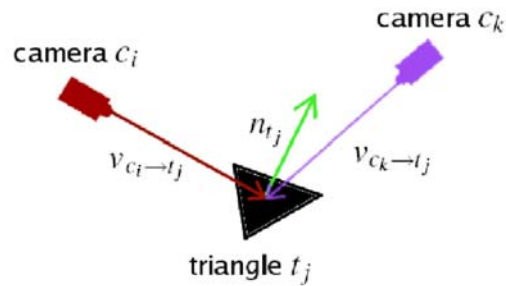
Figure 4.6: A surface triangle, with the considered vectors: normal vector and camera viewing directions

Still, in a per-pixel approach, we do not have the geometrical data. We solve this problem by an additional rendering of the object from the current viewpoint, where we use the polygon ID as its color. Thus, we can determine what polygons are visible from the viewpoint and exactly which pixel of the current image view belongs to which triangle.

Determining if a point viewed by the viewer is occluded or not to the cameras is a less obvious problem and a time consuming one. Methods to determine what points are occluded were briefly presented in the "State of the Art" chapter.

We propose to bypass the occlusion checking by doing a basic statistical test. The strong condition that has to be fulfilled is that for each pixel at least three cameras have to pass the first visibility test, and the majority has to see the correct color.

4.4.2 Eliminating the wrong colors

The method for eliminating the wrong colors is the one described in the previous section, with the exception that, instead of using only the closest 3 or 4 cameras, we use all cameras that face a surface point to determine its final color.

After deciding which colors are accepted, they are blended using the weight function to obtain the pixel color. We define the blending weight of a camera c_i for a point p as:

$$weight(c_i, p) = \begin{cases} cos(v_{c_i \rightarrow t_j}, n_{t_j}) & \text{if } cos(v_{c_i \rightarrow t_j}, n_{t_j}) < 0 \\ 0 & otherwise \end{cases}$$

where t_j is the triangle to which the point p belongs, and n_{t_j} is the normal to the triangle. We take in consideration the cosine because computing the angle requires an inverse cosine operation, and it basically amounts to the same weight. Similarly to the first method, the weight function is summed to 1 by the formula:

$$\overline{weight}(c_i, p) = \frac{weight(c_i, p)}{\sum_{j=1}^{n} weight(c_j, p)} \qquad \text{where } n \text{ is the number of cameras}$$

4.4.3 Algorithm

The resulting algorithm for the described method is the one presented in Algorithm 4.4.1.

Unlike the view-dependent method, this method results in a texture that doesn't change when the viewing camera is moved. The front-facing triangles check and the dominant color selection help in eliminating the errors in texturing. It is a clearer and more accurate result than the one obtained by the view-depending method.

4.4.4 Median filter

The view-independent method greatly depends on the model's normals, which change from one frame to another. This amounts to color variation in time, lessened by texture blending, but still visible on the moving triangle boundaries. We tried to reduce this variation without specifically following the pixel color in time.

To do this, we applied an image filter to the 2D output of the view-independent method. Because the purpose of the filter is to eliminate jumps in pixel coloring while preserving the details, we used the median filter, whose main property is that it eliminates outliers without creating new pixels values.

Median filtering considers each pixel in the image and looks at the nearby neighbors to compute the median of their values. It then replaces the pixel color with

Input: A triangulated 3D mesh in the form of successive XYZ triplets
describing the object surface.
Color images and camera parameters for each of the 6 cameras.
**Output**: A textured 3D model.

1 Find out the (partially) visible polygons from the current viewpoint;
2 **for** *each pixel in the image view* **do**
3   **for** *each camera* **do**
4   **if** *the polygon that colored the pixel faces the camera* **then**
5   retain the corresponding color;
6   **end**
7   **end**
8   **if** *there are three or more colors* **then**
9   Compute the mean and standard deviation;
10   **for** *each individual color* **do**
11   If it is not in the allowed interval, exclude;
12   **end**
13   **if** *there are colors left* **then**
14   Compute the weighted mean;
15   **end**
16   **else**
17   Compute the color using neighboring colors
18   **end**
19   **end**
20   **if** *there are two colors* **then**
21   Compute the weighted mean;
22   **end**
23   **if** *there is only colors* **then**
24   retain that color;
25   **end**
26   **if** *there is no color* **then**
27   Compute the color using neighboring colors;
28   **end**
29 **end**
30 Draw;

Algorithm 4.4.1: View independent algorithm with visibility check and selection of the dominant color

the middle pixel value. This filter eliminates unrepresentative pixels, smooths the transition between close colors and preserves the sharp, clearly defined edges (see Figure 4.7).

Still, median filtering slows down the algorithm considerably and, if it preserves the details, it nevertheless produces a slightly unrealistic texture.
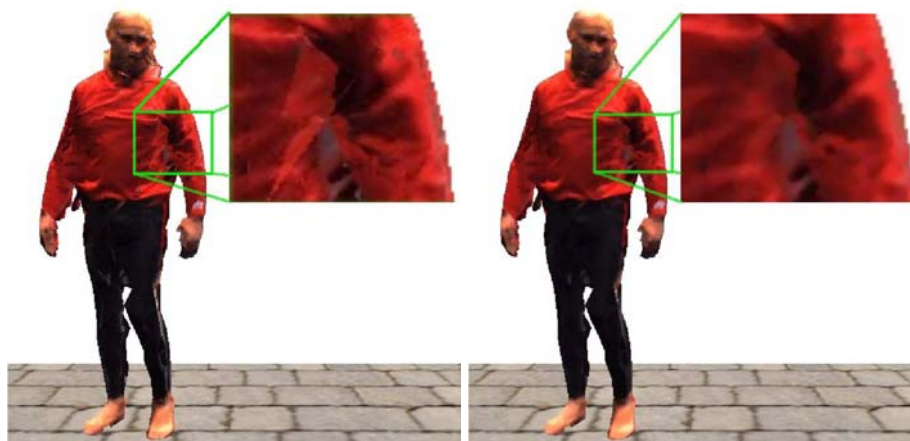
(a) Our view-independent method without using the median filter.

(b) The view-independent method with median filter. Note the color changing more smoothly with the second method, especially in the portion marked by the green rectangle.

Figure 4.7: Comparison between textures before and after using the median filter.

Even if the view-independent method succeeds in texturing well most of the times, it fails for the points that are invisible for all cameras, but front facing some of them. A glaring example can be seen in Figure 4.8.

For these kind of problems, we have to determine the occluded 3D points.

4.5 Shadow mapping method

This method uses the shadow-mapping technique to determine, for each camera, the occluded points. It then employs the view-independent mode of texturing to combine color information taken from all the cameras than see a point. In order to eliminate errors derived from point – eye view and point – camera view distance computations, the dominant color is chosen. For small invisible portions, median filtering is used to fill in the color.

4.5.1 Shadow Mapping

Shadow mapping, first introduced in [22], is a method for producing the shadows cast by objects present in a scene.
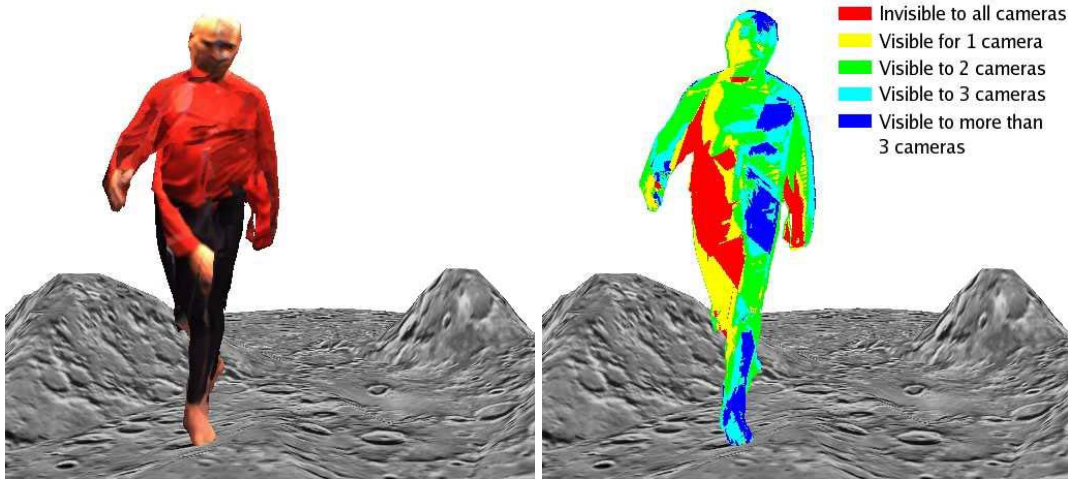
Figure 4.8: a) An example of bad-textured model, when a large portion is invisible for all cameras (see the hand projected on the body). b) The same model, colored in accordance with the degree of visibility.

To do this, it first represents the scene from the light's point of view, and stores the distances to the closest pixels; the result is a "depth map" or "shadow map" texture.

It then renders the scene for the second time, from the eye's point of view. For each pixel, it determines the (X, Y, Z) position relative to the light, and compares the depth value D at the (X, Y) position in the shadow map with the Z value of the pixel. If $Z \cong D$, then the pixel is "viewed" by the light, and therefore illuminated. If $Z > D$, there is something closer to the light than the considered pixel, and the point is shadowed. Figure 4.9 illustrates the described process for both cases.

We have chosen this technique to determine the occluded points because it is faster than methods based on geometrical computations, such as the shadow volumes. It is an image space technique that does not require additional knowledge or processing of the scene geometry, so it's computational time doesn't scale with the increase of geometrical information. Moreover, the depth map can be stored in a texture format, and therefore be more easily combined with the texture images we work with.

We use this method to check, for each pixel belonging to the eye view image plane, whether it is seen by a camera or not. We do this by considering each camera as a light source and computing the lit areas. These pixels are exactly those visible to the camera, and therefore the camera image can be used in texturing them. A scene example, with its depth map, computed shadows and the resulting texturing can be viewed in Figure 4.10.
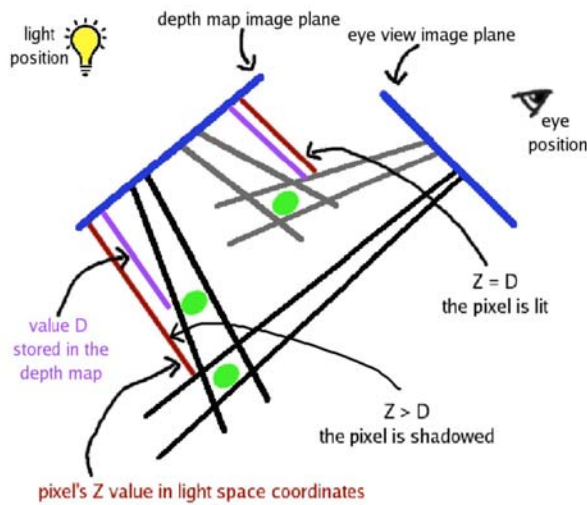
Figure 4.9: Depth comparisons that occur in shadow mapping (image inspired from Mark Kilgard's presentation [12]).

4.5.2 Eliminating the wrong colors

The shadow map method is prone to errors, especially at the border between the visible and occluded parts. This is because when the geometry is rasterized from eye's point of view, it will be sampled in different locations than when it is rasterized from the light's point of view. As a result, invisible pixels will be considered visible, and "leaks" in texturing will result in erroneous final colors. Part of these pixels are eliminated by considering only the front-facing triangles, but problems at the boundary line between visible and invisible regions still remain. An example is shown in Figure 4.11.

Moreover, the 3D model used during this internship is not a perfect one, and this induces errors in texturing even after determining the occluded parts. You can see in Figure 4.12 how a small "bump", present in the model but not in the real object, is considered as belonging to the hand, and textured as such, when in fact it belongs to the torso.

This sort of errors can be eliminated by comparing the colors taken from all cameras used to determine the final color. So, it is still necessary to apply the standard deviation test to the available colors.

After eliminating the wrong colors, we compute the weighted mean as in section 4.4.2 on page 33.

Another problem is the small regions that are invisible to all cameras. They are apparent when using the shadow-mapping method, because no texture is mapped on them. For these regions, we look at the already colored neighboring pixels, and we compute the median color; this color we assign to the invisible pixel.
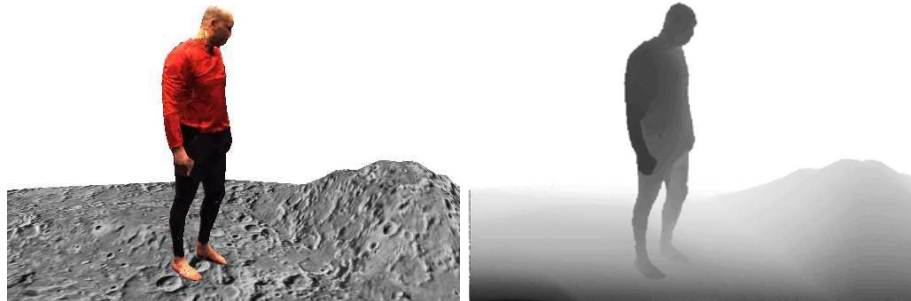
(a) Scene viewed from the light's point of view

(b) The corresponding depths in gray levels. Darker values are closer than the light ones.
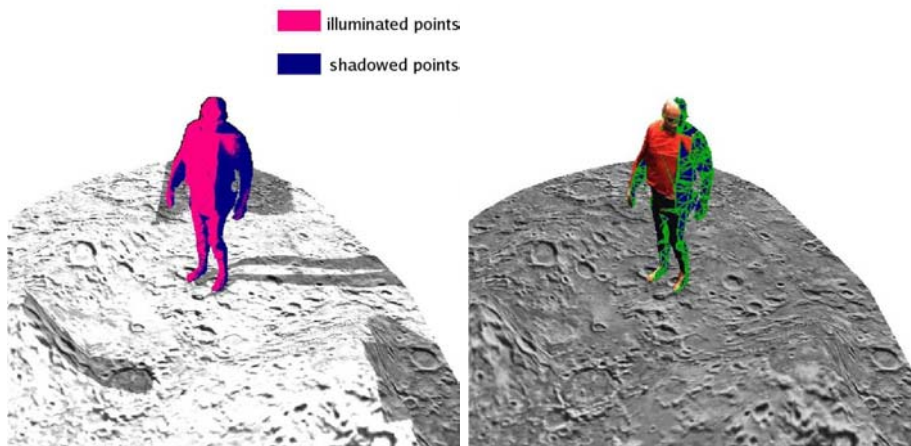
(c) Scene with the resulting shadows

(d) The 3D wireframe model with texture applied using the shadow map

Figure 4.10: Depth map example.

The altered median filter is also useful for eliminating the borders appearing at texture intersection. This regions are generally not agreed on by the cameras that see them, meaning that the colors are too far apart, and we can detect this during the standard deviation test. For this regions we also choose the median color of the neighbors.

Photos presenting the median filter results can be seen in Figure 4.13.

(a) Texture result after shadow mapping for one camera.

(b) Close-up on the texture. See the errors towards the texture border.
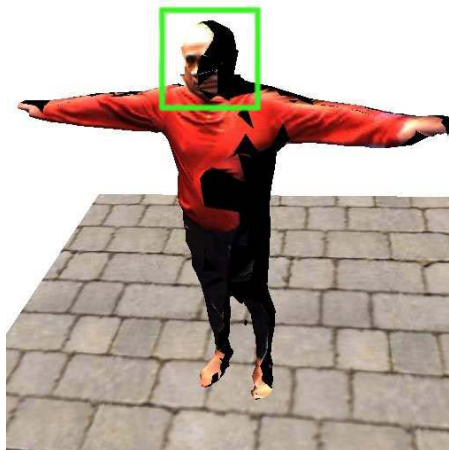
(c) the same texture, after the elimination of back-facing triangles. Note that the border is clearer.

Figure 4.11: Shadow mapping imperfections, partly rectified by checking for front-facing polygons.

4.5.3 Choice of cameras

One idea to determine what cameras are of no use for the current viewpoint is to count how many of the visible triangles face each camera. We know the visible triangles by assigning to each a different color, representing the scene once and
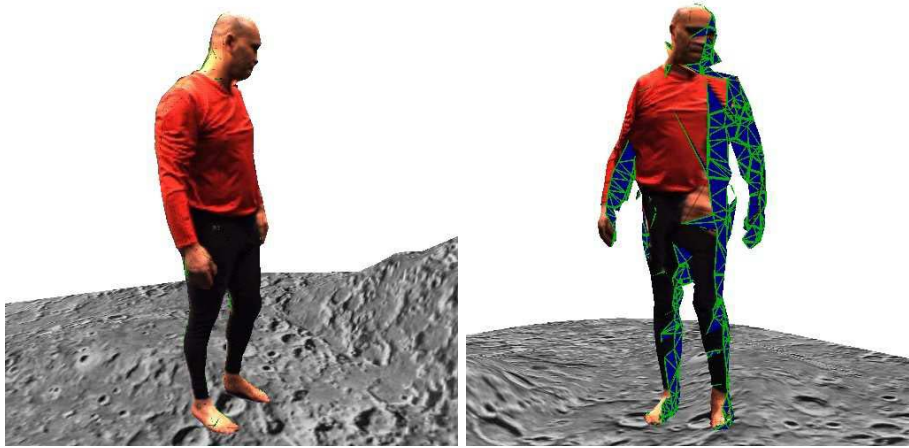
(a) Model viewed from the camera's point
of view; note the normal looking left hand

(b) Model viewed from a different angle;
note the left hand textured on the torso
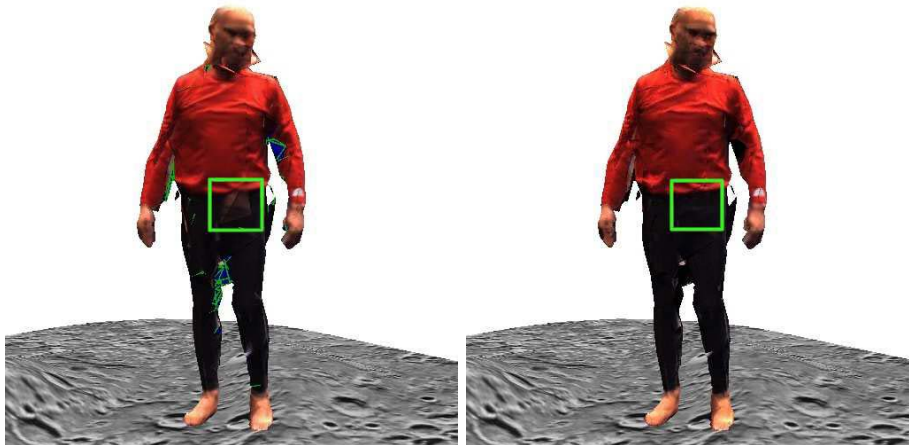
(c) Model textured using all the cameras;
the hand "shadow" is still present

(d) Model after choosing the dominant
color; the hand shadow is gone

Figure 4.12: Model imperfections reflected in the texturing.

see what colors are present (as described in section 4.4.1 on page 31). Then, for
each camera c_i and each visible triangle t_j, we compute $d = n_{t_j} \cdot v_{c_i \to t_j}$, where n_{t_j}
is the triangle normal vector and $v_{c_i \to t_j}$ is the viewing direction from c_i towards the
centroid of t_j (see Figure 4.6). If $d < 0$, then the triangle is facing the camera. We
finally consider only cameras that have mode then one front-facing triangle.

This method, although it eliminates one or two cameras for each frame, takes as
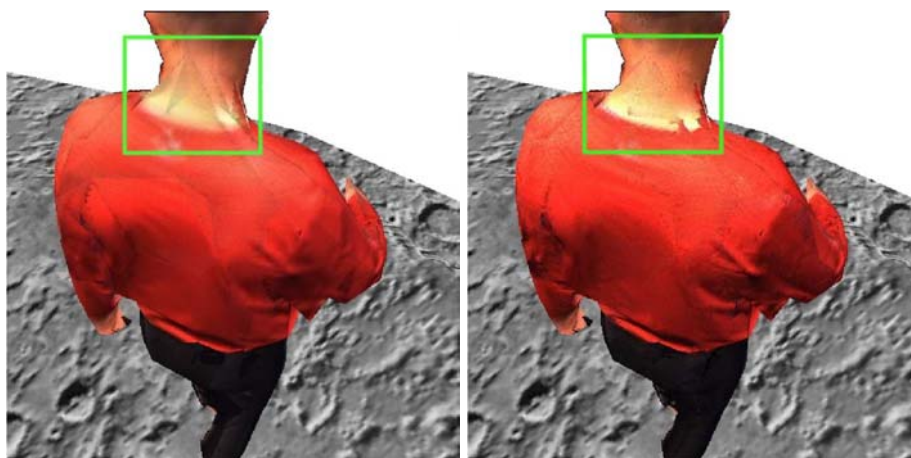
(a) Model textured using shadow maps; note the borders on the neck and shoulders

(b) Model after applying the median filter

(c) Model textured using shadow maps; blue and green wireframe can be seen in several places

(d) Same model after applying the median filter

Figure 4.13: Snapshots comparing the shadow mapping algorithm with and without the median filter

much time to compute as it would take to consider the eliminated cameras, so it is not a good solution.

Another idea was to eliminate the camera that observes the scene from an angle farthest away from the current one. To do this, we consider the center of the scene to be the center of the model's bounding volume and we compute the angle with which each camera deviates from the current viewpoint (see section 4.3.2 on

page 29). Even if not a very exact measure, this method succeeds in quickly choosing the right camera, and there are not perceptible differences between it and using all cameras. Moreover, the speed is visibly increased.

4.5.4   Algorithm

The algorithm runs as described in Algorithm 4.5.1.

1 Retain only the 5 cameras closest to the viewing point;
2 Find out the (partially) visible polygons from the current viewpoint;
3 for *each considered camera* **do**
4   Compute the shadow map; Retain only the illuminated pixels that belong to the front facing polygons;
5 end
6 for *each pixel in the image view* **do**
7   **for** *each considered camera* **do**
8   **if** *the polygon that colored the pixel faces the camera* **then**
9   retain the corresponding color;
10   **end**
11   **end**
12   **if** *there are three or more colors* **then**
13   Compute the mean and standard deviation;
14   **for** *each individual color* **do**
15   If it is not in the allowed interval, exclude;
16   **end**
17   **if** *there are colors left* **then**
18   Compute the weighted mean;
19   **end**
20   **else**
21   Compute the color using neighboring colors
22   **end**
23   **end**
24   **if** *there are two colors* **then**
25   Compute the weighted mean;
26   **end**
27   **if** *there is only colors* **then**
28   retain that color;
29   **end**
30   **if** *there is no color* **then**
31   Compute the color using neighboring colors;
32   **end**
33 end
34 Draw;

Algorithm 4.5.1: View independent algorithm using shadow maps and selection of the dominant color

Chapter 5

Results and Discussions

5.1 Results

All our results were obtained with the system used by the CYBER-II project. The system has 6 cameras, distributed as seen in Figure 2.1. The algorithms were tested on a computer having an Intel 2.40GHz CPU and a GeForce 6800 GT graphic card.

We set the resolution of the rendered novel view to 512x512, and we tried all the algorithms for a model of about 5000 polygons. We looked for a texturing method that comes close to the real object aspect and hasn't many errors, while working in real time.

The results, characterized by the visual quality of the video and by the frame-rate, are presented in Table 5.1. The visual quality of different algorithms is difficult to compare and the comments may be subjective. Also difficult is to show video characteristics by still images, as we did in this paper.

5.2 Discussion

The view-dependent method, combined with the standard deviation test, succeeds in eliminating a great deal of the wrong colors. Still, regions visible to less then three cameras are not considered by the standard deviation test, and errors do occur. Besides, this method, although fast, needs a greater number of cameras to be able to smoothly interpolate between them. In our system, the cameras are relatively far apart, and the passing from one camera view to another is visible to the user.

The view-independent method results in a "fixed" texture, where the pixel color doesn't change with the change of viewpoint. For the regions where the object is seen by at least 2 cameras, the algorithm succeeded in eliminating the wrong colors and in seamlessly mixing data from various cameras. However, for the parts where the object is seen by one camera or none at all, the algorithm works without color

Texturing method	Visual quality	Speed (in frames per second)
View dependent algorithm	Acceptable. Model imperfections give place to errors such as background being projected on the object.	42 fps
View independent algorithm	Acceptable. Clear texturing, but invisible portions wrongly colored.	27 fps
View independent algorithm with median filter	Acceptable. No fast-color variation, in space or time.	11 fps
Shadow mapping	Good. Invisible regions detected. Texture junction visible.	24 fps
Shadow mapping with median filter	Acceptable. Tends to accentuate the difference at texture junction.	11 fps
Shadow mapping, dominant color, median filter for invisible pixels. Use all 6 cameras	Good. Most errors are detected and eliminated. Invisible regions are properly filled.	17 fps
Shadow mapping, dominant color, median filter for invisible pixels. Use the best 5 cameras	Good. Most errors are detected and eliminated. Invisible regions are properly filled.	23 fps

Table 5.1: Algorithm results

elimination; this means that the texture that passes through the occluding geometry is not detected. The median filter variation of the algorithm smooths the texture and prevents the fast change of color in time, but it doubles the computation time.

Shadow mapping method, with dominant color selection and median filter for invisible pixels, has hardly any errors for the regions seen by the 4 front cameras. It decides correctly the color for small invisible regions and eliminates most of the texture boundaries. For the back views, some errors are noticeable at the junction between the 2 camera views; they are due to insufficient color information at the intersection.

Chapter 6

Conclusions and Future work

6.1 Conclusions

We implemented a number of per-pixel algorithms for multi-view texture mapping. We used them to texture polyhedron-based 3D models obtained by computing the visual hull from images taken by six video cameras.

We introduced the idea of eliminating the wrong colors by doing a simple standard deviation test. This has very good results in correcting the "leaks" in visibility test and in removing artefacts created by the use of an imperfect model. Moreover, this test helps in detecting the texture frontiers, in order to rectify the color difference visible at those points.

We used a modified version of the median filter to fill the color gaps for the small invisible regions. The proposed method works in the image plane and doesn't need the geometrical information that Gouraud interpolation requires.

The view-independent method we implemented, coupled with the dominant color choice, succeeds in eliminating wrong colors for pixels viewed by more that two cameras, without doing a time-consuming occlusion checking. However, the six-camera system we used does not offer sufficient information for this algorithm to work without error. There are frequent cases were only two cameras, or less, see a particular point.

The algorithm that uses shadow-mapping visibility check is better adapted for our system. Together with the standard deviation test and the hole-filling method, it has as result a close-to-reality texture. This is an algorithm suited to a small number of available cameras. Since it depends on the number of lights supported by the OpenGL implementation, it is restricted to a maximum of eight cameras. For more cameras, the view-independent algorithm would probably be able to correctly texture an object without recourse to the shadow-map occlusion checking.

6.2 Future Work

Further enhancements are both necessary and feasible. Thus, a hardware-implementation should be considered, since the main time-consuming task in our algorithm is transferring information from the frame-buffer to the CPU. Determining the frontier triangles that appear in more than one camera view and making a local color correction might also be of interest.

Moreover, we would like to consider a continuity in time of the computed pixel colors. The definition of a framework for measuring the video quality of different texturing algorithms could be another direction of research.

An increase in the number of cameras should help in obtaining a better model and would also give more information to be used in texturing.

Appendix A

Article in the RoCHI Conference Proceedings

Alexandrina Orzan and Jean-Marc Hasenfratz. Omnidirectional texturing of human actors from multiple view video sequences. In *Conference on Computer-Human Interaction*, 2005.

Omnidirectional texturing of human actors from multiple view video sequences

Alexandrina Orzan,[*] Jean-Marc Hasenfratz[†]

Artis,[‡] GRAVIR/IMAG - INRIA

Abstract

In 3D video, recorded object behaviors can be observed from any viewpoint, because the 3D video registers the object's 3D shape and color. However, the real-world views are limited to the views from a number of cameras, so only a coarse model of the object can be recovered in real-time. It becomes then necessary to judiciously texture the object with images recovered from the cameras. One of the problems in multi-texturing is to decide what portion of the 3D model is visible from what camera. We propose a texture-mapping algorithm that tries to bypass the problem of exactly deciding if a point is visible or not from a certain camera. Given more than two color values for each pixel, a statistical test allows to exclude outlying color data before blending.

1 Introduction

Currently, visual media such as television and motion pictures only present a 2D impression of the real world. In the last few years, increasingly more research activity has been devoted to investigate 3D video from multiple camera views. The goal is to obtain a free-viewpoint video, where the user is able to watch a scene from an arbitrary viewpoint chosen interactively.

The possible applications are manifold. A free-viewpoint system can increase the visual realism of telepresence technology [1], thus enabling users in different locations to collaborate in a shared, simulated environment as if they were in the same physical room. Also, special effects used by the movie industry, such as *freeze-and-rotate* camera, would be made accessible to all users.

For free-viewpoint video, a scene is typically captured by N cameras. From the views obtained by the cameras a 3D video object, with its shape and appearance, is created. The shape can be described by polygon meshes, point samples or voxels. In order to make the model more realistic, appearance is typically described by the textures captured from the video streams. Appearance data is mapped onto the 3D shape, thus completing the virtual representation of the real object. The 3D video object can be seamlessly blended into existing content, where it can be interactively viewed from different directions, or under different illumination.

Since people are central to most visual media content, research has been dedicated in particular to the extraction and reconstruction of human actors.However, the system used in this article is not restricted to human actors, as [2]. Moreover, it allows the acquisition of multiple objects present in the scene.

The rest of the paper proceeds with a review of related work in section 2. Section 3 will be dedicated to describing the proposed method of texture-mapping, after which results and future tasks are discussed.

2 Previous Work

Over the last few years, several systems with different model reconstruction and different ways of texturing the 3D model have been proposed.

---

[*]ENS de Cachan - France

[†]University Pierre Mendès France - Grenoble II

[‡]Artis is a team of the GRAVIR/IMAG laboratory, a joint research unit of CNRS, INPG, INRIA, UJF

[1]"Telepresence technology" enables people to feel as if they are actually present in a different place or time (S. Fisher & B. Laurel, 1991) or enables objects from a different place to feel as if they are actually present (T. Lacey & W. Chapin, 1994).

2.1 3D Model reconstruction

Two different approaches of model reconstruction have been studied in the recent years: model-free and model-based reconstruction.

Model-free reconstruction makes no a priori assumptions on scene geometry, allowing the reconstruction of complex dynamic scenes. In human modeling it allows the reproduction of detailed dynamics for hair and loose clothing.

Most model-free methods aim to estimate the visual hull, an approximate shell that envelopes the true geometry of the object [10]. To achieve this, object silhouettes are extracted from each camera image by detecting the pixels not belonging to the background.

The visual hull can then be reconstructed either by voxel-based or polyhedron-based approaches. The first approach discretizes a confined 3D space in voxels and carves away those voxels whose projection fall outside the silhouette of any reference view [7]. Polyhedron-based approaches represent each visual cone as a polyhedral object and computes the intersection of all visual cones [11, 14, 13].

The visual hull allows real-time reconstruction and rendering, yet it needs a large number of views to accurately represent a scene, otherwise the obtained model is not very exact.

Model-based reconstruction assumes that the real object is a human body and uses a generic humanoid model, which is deformed to fit the observed silhouettes [2, 8, 9]. Although it results in a more accurate model and permits motion tracking over time, this approach is restricted to a simple model and does not allow complex clothing movements. Moreover, it places a severe limitation on what can be captured (i.e. a single human body) and it is not real-time.

In this paper, the 3D model used is the one created in the context of CYBER-II project[2], a polyhedron-based model obtained in real-time.

2.2 Multi-view texture mapping

Original images from multiple viewpoint are often mapped onto recovered 3D geometry in order to achieve realistic rendering results [3]. Proposed methods for multi-texture mapping are either view-dependent or view-independent.

View-dependent texture mapping considers only the camera views closest to the current viewpoint. In between camera views, two to four textures are blended together in order to obtain the current view image [4, 5]. This method exhibits noticeable blending artifacts in parts where the model geometry does not exactly correspond to the observed shape. What's more, the result is usually blurred and the passing from one camera view to another does not always go unnoticed.

View-independent texture mapping selects the most appropriate camera for each triangle of the 3D model, independently of the viewer's viewpoint [2, 8, 13]. The advantage of this method is that it does not change the triangle texture when the user changes the viewpoint. Moreover, the blurred effect is less noticeable. However, the problem is that the best camera is not the same from patch to patch, even if they are neighboring. Here also, blending between visible views is necessary in order to reduce the abrupt change in texture at triangle edges.

Blending is done using various formulas that depend of:

- the angle between the surface normal and the vector towards the considered camera

- the angle between the surface normal and the vector towards the viewpoint

- the angle the vector towards a camera and the vector towards the viewpoint

Blending weights can be computed per vertex or per polygon [2, 4, 5]. Matsuyama [13] proposes using this method for determining each vertex color and then paints the triangles with the colors obtained by linearly interpolating the RGB values of its vertices. However, for large triangles, small details like creases in the clothes are lost.

Li and Magnor [12] compute the blending for each rasterized fragment, which results in a more accurate blending.

2.3 Visibility

Visibilities with respect to reference views are very important for multi-view texture mapping. For those parts that are invisible in a reference view, the corresponding color information should be ignored when blending multiple textures.

---

[2]http://artis.imag.fr/Projects/Cyber-II/

Debevec et al. [3] splits the object triangles so that they are either fully visible or fully invisible to any source view. This process takes a long time even for a moderately complex object and is not suitable for real-time applications. Matusik [14] proposes computing the vertex visibility at the same time that the visual hull is generated. Magnor et al. [12] solves the visibility problem per fragment, using shadow mapping. However, they require rendering the scene from each input camera viewpoint and is not real-time even with a hardware-accelerated implementation.

We propose a per pixel method that checks only polygon visibility and eliminates the wrong colors by considering only those colors that are close to a computed average.

3 Texture mapping algorithm

3.1 Model constraints

The polyhedron-based model-free method recreates the geometrical object at each frame. The number of polygons, their form and position in space vary greatly in time, so we cannot track vertices from one frame to another.

This means that it is impossible to decide the color of the polygons only once, at the beginning of the video. Color values have to be computed in real-time, for each frame.

3.2 Algorithm description

To achieve realistic rendering results, we use the projective texture mapping, a method introduced by Segal [15] and included in the OpenGL graphics standard. But the current hardware implementation of projective texture mapping in OpenGL lets the texture pass through the geometry and be mapped onto all back-facing and occluded polygons. Thus it is necessary to perform visibility check so that only polygons visible to a particular camera are texture mapped with the corresponding image.

A point p on the object's surface is visible from a camera c_i if (1) the triangle t_j to which the point belongs faces the camera and (2) the point is not occluded by any other triangles.

The first condition can be fast determined by checking the equation $n_{t_j} \cdot v_{c_i \to t_j} < 0$, where n_{t_j} is the triangle normal vector and $v_{c_i \to t_j}$ is the viewing direction from c_i towards the centroid of t_j.

Still, in a per-pixel approach, we do not have the geometrical data. We solve this problem by an additional rendering of the object from the current viewpoint, where we use the polygon ID as its color. Thus, we can determine what polygons are visible from the viewpoint and exactly which pixel of the current image view belongs to which triangle.

Determining if a point viewed by the viewer is occluded or not to the cameras is a less obvious problem. Methods to determine what points are occluded were briefly presented in the previous section. We propose to bypass the occlusion checking by doing a basic statistical test. The strong condition that has to be fulfilled is that for each pixel at least three cameras have to pass the first visibility test, and the majority has to see the correct color. Still, this is usually the case with a system having an evenly distributed camera configuration.

As all the cameras are calibrated prior to use and the images are acquired at the same time and in the same lighting conditions, we can compare colors and calculate distances in the RGB space [1, 6].

If a sufficient number of color values are available for a pixel, we compute the mean (μ) and the standard deviation (τ) for each of the R, G, B channels. Individual colors falling outside the range $\mu \pm \beta \cdot \tau$ for at least one channel are excluded. The factor β permits us to modify the confidence interval for which the colors are accepted. The classical normal deviation test considers β is 1. We experimentally concluded that it was best to set it at 1.17, to allow for slight errors in manipulating the color-values.

If less than three possible colors are available for a pixel, we do not exclude any of them.

A weighted mean of all contributing images is finally used for texturing each particular pixel. The blending weight is computed using the value of the $\cos(angle(n_{t_j}, v_{c_i \to t_j}))$.

If the pixel is invisible for all cameras, we compute its color using the color values of the neighbours whose color was already decided.

The algorithm runs as follows:

```
 1: for all polygons in the 3D model do
 2:    check if they are at least partially visible from the
       current view
 3: end for
 4: for all pixels in the image view do
 5:    for all cameras do
 6:       if the polygon that colored the pixel faces the
          camera then
 7:          retain the corresponding color
 8:       end if
 9:       if there are three or more colors then
10:          compute the mean and standard deviation
11:          for all colors do
12:             if they are not in the allowed interval
                then
13:                exclude
14:             end if
15:          end for
16:          compute the weighted mean
17:       else if there are two colors then
18:          compute the weighted mean
19:       else if there is no color then
20:          compute the color using neighbouring col-
             ors
21:       end if
22:    end for
23: end for
24: draw
```

4   Results

We tested this algorithm with the system used by the CYBER-II project. The system has 6 cameras, 4 in the front and 2 in the back, as seen in Figure 1[3].

For the front views, the algorithm succeeded in eliminating the wrong colors and in seamlessly mixing data from various cameras. Moreover, the pixel color doesn't change with the change of viewpoint. Images comparing view-dependent and view-independent algorithms, without occlusion checking, and our method can be seen in Figure 2.

However, for the back views, where the object is seen by at most 2 cameras, the algorithm does only a weighted average, without color elimination.

---

[3]video sequences were acquired with the Grimage platform of Inria Rhône-Alpes

Figure 1: Camera setting

Figure 2: a) View dependent, b) View independent, c) Our method

We set the resolution of the rendered novel view to 512x512, and we tested the algorithm for a model of about 5000 polygons. On a Intel 2.40GHz CPU and a GeForce4 Ti 4800 graphic card, the frame rate is of 17 fps.

5   Conclusions and Future work

A per-pixel algorithm for multi-view texture mapping has been implemented. It succeeds in eliminating wrong colors for pixels viewed by more that 2 cameras, without doing a time-consuming occlusion checking.

Yet, further enhancements are both necessary and feasible. Thus, a hardware-implementation should be considered, since the main time-consuming task in our algorithm is transferring information from the framebuffer to the CPU. Moreover, we would like to consider a continuity in time of the computed pixel colors and a dynamic deactivation of the unused cameras.

References

[1] A. Agathos and R. Fishe. Colour texture fusion of multiple range images. In *Proceedings of the 4th International Conference on 3-D Digital Imaging and Modeling*, pages 139– 146, 2003.

[2] Joel Carranza, Christian Theobalt, Marcus Magnor, and Hans-Peter Seidel. Free-viewpoint video of human actors. *ACM Trans. on Computer Graphics*, 22(3):569 – 577, July 2003.

[3] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics*, 30(Annual Conference Series):11–20, 1996.

[4] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Workshop on Rendering*, 1998.

[5] Bastian Goldlücke and Marcus Magnor. Real-time microfacet billboarding for free-viewpoint video rendering. In *Proceedings of ICIP 2003, IEEE Computer Society*, volume 3, pages 713–716, 2003.

[6] L. Grammatikopoulos, I. Kalisperakis, G. Karras, T. Kokkinos, and E. Petsa. On automatic orthoprojection and texture-mapping of 3d surface models. In *ISPRS Congress - Geo-Imagery Bridging Continents*, 2004.

[7] Jean-Marc Hasenfratz, Marc Lapierre, and François Sillion. A real-time system for full body interaction. *Virtual Environments*, pages 147–156, 2004.

[8] Adrian Hilton and Jonathan Starck. Model-based multiple view reconstruction of people. In *IEEE International Conference on Computer Vision*, pages 915–922, 2003.

[9] Adrian Hilton and Jonathan Starck. Multiple view reconstruction of people. In *3D Data Processing, Visualization, and Transmission*, pages 357–364, 2004.

[10] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):150–162, 1994.

[11] Ming Li, Marcus Magnor, and Hans-Peter Seidel. Online accelerated rendering of visual hulls in real scenes. In *Journal of WSCG*, 2003.

[12] Ming Li, Marcus Magnor, and Hans-Peter Seidel. A hybrid hardware-accelerated algorithm for high quality rendering of visual hulls. In *Proceedings of the 2004 conference on Graphics interface*, pages 41–48, 2004.

[13] Takashi Matsuyama and Takeshi Takai. Generation, visualization, and editing of 3d video. In *3D Data Processing, Visualization, and Transmission*, page 234, 2002.

[14] Wojciech Matusik, Chris Buehler, and Leonard McMillan. Polyhedral visual hulls for real-time rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 115–126, 2001.

[15] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 249–252, 1992.

Bibliography

[1] A. Agathos and R. Fishe. Colour texture fusion of multiple range images. In *Proceedings of the 4th International Conference on 3-D Digital Imaging and Modeling*, pages 139–146, 2003.

[2] Joel Carranza, Christian Theobalt, Marcus Magnor, and Hans-Peter Seidel. Free-viewpoint video of human actors. *ACM Trans. on Computer Graphics*, 22(3):569–577, 2003.

[3] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics*, 30(Annual Conference Series):11–20, 1996.

[4] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Workshop on Rendering*, 1998.

[5] Bastian Goldlücke and Marcus Magnor. Real-time microfacet billboarding for free-viewpoint video rendering. In *Proceedings of ICIP 2003, IEEE Computer Society*, volume 3, pages 713–716, 2003.

[6] L. Grammatikopoulos, I. Kalisperakis, G. Karras, T. Kokkinos, and E. Petsa. On automatic orthoprojection and texture-mapping of 3d surface models. In *ISPRS Congress - Geo-Imagery Bridging Continents*, 2004.

[7] Jean-Marc Hasenfratz, Marc Lapierre, Jean-Dominique Gascuel, and Edmond Boyer. Real-time capture, reconstruction and insertion into virtual world of human actors. In *Vision, Video and Graphics*, pages 49–56. Eurographics, Elsevier, 2003.

[8] Jean-Marc Hasenfratz, Marc Lapierre, and François Sillion. A real-time system for full body interaction. *Virtual Environments*, pages 147–156, 2004.

[9] Adrian Hilton and Jonathan Starck. Model-based multiple view reconstruction of people. In *IEEE International Conference on Computer Vision*, pages 915–922, 2003.

[10] Adrian Hilton and Jonathan Starck. Multiple view reconstruction of people. In *3D Data Processing, Visualization, and Transmission*, pages 357–364, 2004.

[11] Takeo Kanade, Peter Rander, and P. J. Narayanan. Virtualized reality: Constructing virtual worlds from real scenes. *IEEE MultiMedia*, 4(1):34–47, 1997.

[12] Mark Kilgard. Shadow mapping with today's hardware. In *CESA Developers Conference*, 2001.

[13] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):150–162, 1994.

[14] Ming Li, Marcus Magnor, and Hans-Peter Seidel. Online accelerated rendering of visual hulls in real scenes. In *Journal of WSCG*, 2003.

[15] Ming Li, Marcus Magnor, and Hans-Peter Seidel. A hybrid hardware-accelerated algorithm for high quality rendering of visual hulls. In *Proceedings of the 2004 conference on Graphics interface*, pages 41–48, 2004.

[16] Takashi Matsuyama and Takeshi Takai. Generation, visualization, and editing of 3d video. In *3D Data Processing, Visualization, and Transmission*, pages 234–245, 2002.

[17] Wojciech Matusik, Chris Buehler, and Leonard McMillan. Polyhedral visual hulls for real-time rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 115–126, 2001.

[18] P. J. Narayanan, Peter Rander, and Takeo Kanade. Constructing virtual worlds using dense stereo. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, pages 3–11, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 249–252, 1992.

[20] Christian Theobalt, Joel Carranza, Marcus A. Magnor, and Hans-Peter Seidel. Combining 3d flow fields with silhouette-based human motion capture for immersive video. *Graph. Models*, 66(6):333–351, 2004.

[21] Sundar Vedula, Simon Baker, and Takeo Kanade. Image-based spatio-temporal modeling and view interpolation of dynamic events. *ACM Trans. Graph.*, 24(2):240–261, 2005.

[22] Lance 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.