# Rendu réaliste de surfaces par la méthode de "ray tracing"

Jos Stam
mémoire de licence ès sciences informatiques
Université de Genève

juillet-août 1988

# Table des Matières


**1 Introduction** **5**

**2 La méthode de "ray tracing"** **7**
2.1 Algorithme de base . . . . . 7
2.2 Modèle mathématique . . . . . 8
2.3 Modèle d'illumination . . . . . 9
2.3.1 Equation générale . . . . . 9
2.3.2 Intensité ambiante . . . . . 10
2.3.3 Intensité diffuse . . . . . 10
2.3.4 Intensité spéculaire . . . . . 10
2.3.5 Intensité réfléchie et transmise . . . . . 11
2.3.6 Remarques . . . . . 11

**3 "Ray tracing" de surfaces généralisées** **13**
3.1 Définitions . . . . . 13
3.1.1 Représentation implicite . . . . . 13
3.1.2 Représentation paramétrique . . . . . 14
3.2 Problème de l'intersection . . . . . 14
3.2.1 Surfaces sous forme implicite . . . . . 14
3.2.2 Surfaces sous forme paramétrique . . . . . 15
3.3 Méthodes itératives . . . . . 16
3.3.1 La méthode de Newton . . . . . 16
3.3.2 Choix de la valeur initiale . . . . . 16
3.4 Racines communes de deux polynômes bivariés . . . . . 16
3.5 Racines réelles de polynômes . . . . . 17
3.5.1 Les suites de Sturm . . . . . 17
3.5.2 Isolement des racines . . . . . 17
3.5.3 Recherche de la racine . . . . . 18
3.6 Problème du vecteur normal . . . . . 19
3.6.1 Représentation implicite . . . . . 19
3.6.2 Représentation paramétrique . . . . . 19

4 Implantation 21
4.1 Description du logiciel . . . . . . . . 21
4.1.1 Description globale . . . . . . . . 21
4.1.2 Code propre à un objet . . . . . . . . 21
4.2 Interface avec l'utilisateur . . . . . . . . 23
4.2.1 Spécification de la scène . . . . . . . . 23
4.2.2 Affichage de la scène . . . . . . . . 26
4.3 Résultats . . . . . . . . 26

5 Conclusion 31

A Article 33

B Code source 35



# Liste des Figures

2.1 principe du "ray tracing" . . . . . . . . 8
2.2 vecteurs intervenant dans le modèle d'illumination . . . . . . . . 12

3.1 Intersections de deux courbes implicites . . . . . . . . 15
3.2 Vecteur normal . . . . . . . . 20

4.1 Structure générale du logiciel . . . . . . . . 22
4.2 paramètres de la caméra . . . . . . . . 24
4.3 Image typique de "ray tracing" . . . . . . . . 27
4.4 Calcul de la forme implicite du tore . . . . . . . . 28
4.5 le tore réflectif . . . . . . . . 29
4.6 le tore transparent . . . . . . . . 30

# Chapitre 1

# Introduction

Dans le domaine de la synthèse d'images par ordinateur la méthode de "ray tracing" (souvent traduit par "traçage de rayons" ou encore "suivi de rayons") a acquis une certaine popularité ces dix dernières années. Ceci pour plusieurs raisons, elle est conceptuellement très simple, proche de notre intuition et donne de très bons résultats (haut degré de réalisme), surtout en ce qui concerne la simulation d'effets comme les ombres, la transparence ou la réflection.

Néanmoins cette méthode est très friande en temps machine, à cause des nombreux calculs qui doivent être effectués. Ceci a pour conséquence de limiter la variété d'objets représentés. La plupart des scènes créées par la méthode de "ray tracing" ne contiennent dans la plupart des cas que des objets "simples", tels que les sphères, les cônes ou les plans.

Ces objets cités ne sont en fait que des cas particuliers de surfaces plus générales. On se propose ainsi dans le présent travail d'étudier et d'implanter des algorithmes permettant de visualiser par la méthode de "ray tracing" des surfaces généralisées. On ne retiendra que les algorithmes qui fournissent des résultats théoriquement exactes. Dans l'implantation on devra néanmoins être attentif aux instabilités numériques provenant de la représentation en virgule flottante.

On fournit notamment dans ce travail un nouvel algorithme pour calculer les intersections d'une droite avec une surface de type "spline". L'algorithme est basée sur des résultats de la *théorie de la reduction* en algèbre, et plus particulièrement sur les travaux de Buchberger. En annexe on trouvera l'exposé complet de cette nouvelle approche.

On commencera, dans ce travail, par donner une brève description de la méthode de base de "ray tracing" et du modèle d'illumination utilisé. Puis on fournira les divers algorithmes de recherche d'intersections pour les surfaces sous forme paramétrique et implicite. On finira en décrivant le logiciel dans lequel les divers algorithmes ont été implantés, et en fournissant quelques images générées.

# Chapitre 2

# La méthode de "ray tracing"

## 2.1 Algorithme de base

Le problème est de représenter une scène sur un plan de projection (écran de l'ordinateur) d'un certain point de vue (position de l'observateur), en tenant compte de tous les effets produits par un ensemble de sources lumineuses. La methode de "ray tracing" est une solution satisfaisante à ce problème.

La méthode de "ray tracing" est conceptuellement très simple: imaginons que l'on se place à l'endroit de l'observateur. Puis procurons-nous une plaque en verre, sur laquelle on a tracé un quadrillage, et plaçons la en face de nous de telle sorte que les carrés du quadrillage coïncident avec les "pixels" [1] de l'écran. La couleur que l'on observe à travers un carré est exactement la couleur que l'on doit afficher à la position du "pixel" correspondant.

Plus précisément, suivons le rayon de lumière en sens inverse, de notre oeil à travers la plaque en verre dans la scène, et considérons le premier objet de la scène intercepté par ce rayon. L'intensité nous provenant de ce point d'intersection situé sur la surface de l'objet est obtenue de la manière suivante. Pour chaque source lumineuse on considère une droite reliant la source avec le point d'intersection. Si cette droite intercepte un autre objet, alors la source ne contribue pas à l'intensité (effet d'ombre). Sinon on détermine la contribution de la source à l'intensité totale par une fonction qui dépend de la distance de la source, de sa couleur, du vecteur pointant vers la source, de la direction du rayon et du vecteur normal à la surface de l'objet au point d'intersection.

Si l'objet intercepté est transparent (respectivement réflectif), alors au lieu de s'arrêter, on suit le rayon réfracté (resp. réfléchi). Ainsi on effectue les

[1] "pixel" provient de "picture element" et désigne le plus petit point représentable sur un écran donné

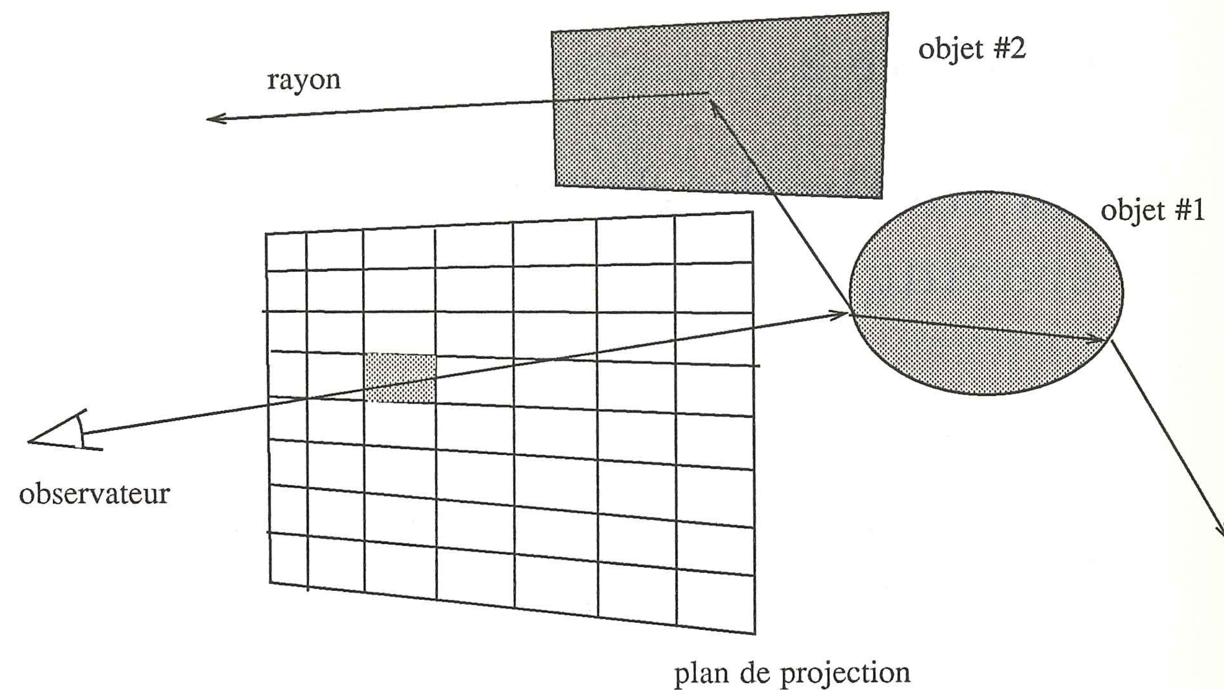


Figure 2.1: principe du "ray tracing"

mêmes étapes décrits plus haut avec ce nouveau rayon. Cette démarche s'arrête lorsqu'un rayon ne touche plus aucun objet ou un objet opaque, dans ce cas on revient en arrière, en ajoutant à chaque fois les intensités venant des rayons réfractés (resp. réfléchis) aux intensités déjà calculées. Une description détaillée sera donnée dans la section 2.3.

## 2.2 Modèle mathématique

Pour pouvoir effectuer les calculs mentionnés dans la section précédente, il faut spécifier tout d'abord la position dans l'espace (par rapport à un référentiel fixé) de l'observateur, du plan de projection, des objets formant la scène et des sources lumineuses.

Chaque rayon est représenté d'une manière paramétrique, c'est-à-dire sous la forme

$$R(t) = R_o + Vt \tag{2.1}$$

où $R_o = (R_o^x, R_o^y, R_o^z)$ est l'origine du rayon, $V = (V^x, V^y, V^z)$ est la direction



du rayon et $t$ est un réel positif indiquant la position sur le rayon. Notons que cette représentation n'est nullement unique, on peut également représenter le rayon, par exemple, comme l'intersection de deux plans non-confondus ou par deux points distincts se trouvant sur le rayon.

Pour chaque objet de la scène il faut fournir une procédure donnant les intersections d'un rayon quelconque avec l'objet ou détectant la non-existance d'une intersection, et une procédure donnant le vecteur normal à la surface de l'objet en tout point.

Prenons pour illustrer ce dernier point l'exemple de la sphère. La sphère est entièrement définie par son centre $(x_o, y_o, z_o)$ et son rayon $R$. Comme tous les points $(x, y, z)$ de la sphère sont à distance $R$ du centre, cela nous donne l'équation:

$$(x - x_o)^2 + (y - y_o)^2 + (z - z_o)^2 - R^2 = 0 \tag{2.2}$$

On cherche les valeurs de $t$ tels que $R(t) = (x, y, z)$ avec $(x, y, z)$ satisfaisant 2.2. On peut reécrire 2.2 comme suit en notations vectorielles avec $\xi_o = (x_o, y_o, z_o)$ et $||(x, y, z)||^2 = x^2 + y^2 + z^2$:

$$||R_o - Vt - \xi_o||^2 - R^2 = 0 \tag{2.3}$$

Ce qui en développant donne une équation polynômiale de degré deux en t:

$$||V||^2 t^2 + 2(V \cdot (R_o - \xi_o))t + ||R_o - \xi_o||^2 - R^2 = 0 \tag{2.4}$$

Il existe donc une procédure pour déterminer les points d'intersection entre un rayon et la sphère. Le vecteur normal $\vec{N}$ en un point $(x, y, z)$ se calcule comme suit:

$$\vec{N} = (\frac{(x - x_o)}{R}, \frac{(y - y_o)}{R}, \frac{(z - z_o)}{R}) \tag{2.5}$$

La sphère est par conséquent un objet que l'on peut visualiser à l'aide de la méthode de "ray tracing".

## 2.3 Modèle d'illumination

On donne ici la procédure permettant de déterminer l'intensité d'un point de la surface d'un objet. Remarquons que la procédure donnée est une des plus simples et qu'il en existe beaucoup d'autres, on peut notamment les trouver dans les ouvrages [9] et [13].

### 2.3.1 Equation générale

L'intensité $I$ que reçoit un observateur d'un point de la surface d'un objet est la contribution de 5 effets: l'intensité ambiante $I_a$ (intensité en l'absence de sources lumineuses), l'intensité diffuse $I_d$ (ne dépend que de la position des sources), l'intensité spéculaire $I_s$ (dépend de la position de l'observateur), l'intensité

réfléchie $I_r$ (dans le cas d'un objet réflectif) et l'intensité transmise $I_t$ (dans le cas d'un objet transparent). Ce qui s'exprime par l'équation:

$$I = I_a + I_d + I_s + I_r + I_t \tag{2.6}$$

Voyons plus en détails comment chaque terme est évalué.

### 2.3.2 Intensité ambiante

L'intensité ambiante $I_a$ est donnée par:

$$I_a = k_a C_s \tag{2.7}$$

où $k_a$ est le coefficient d'intensité ambiante de l'objet et $C_s$ est la couleur de la surface de l'objet.

### 2.3.3 Intensité diffuse

L'intensité diffuse $I_d$ est donnée par la loi de Lambert, ce qui veut dire que pour $N_s$ sources lumineuses on a:

$$I_d = k_d C_s \sum_{j=1}^{N_s} I_j(\vec{N} \cdot \vec{L_j}) \tag{2.8}$$

où $k_d$ est le coefficient de réflection diffuse de l'objet, $I_j$ est l'intensité de la $j$-ième source lumineuse, $\vec{N}$ est le vecteur normal à la surface de l'objet et $\vec{L_j}$ est le vecteur en direction de la $j$-ième source lumineuse.

### 2.3.4 Intensité spéculaire

L'intensité spéculaire $I_s$ est donnée par

$$I_s = k_s C_r \sum_{j=1}^{N_s} I_j(\vec{N} \cdot \vec{H_j})^n \tag{2.9}$$

où $k_s$ est le coefficient de réflection spéculaire de l'objet, $C_r$ est la couleur réflective de la surface de l'objet, l'entier naturel $n$ donne la taille de la tâche spéculaire sur l'objet et $\vec{H_j}$ est le vecteur dans la direction à moitié entre l'observateur et la $j$-ième source lumineuse:

$$\vec{H_j} = \frac{\vec{L_j} - \vec{V}}{||\vec{L_j} - \vec{V}||} \tag{2.10}$$



### 2.3.5 Intensité réfléchie et transmise

L'intensité réfléchie $I_r$ est donnée par:

$$I_r = k_s M \tag{2.11}$$

où $M$ est l'intensité provenant de la direction $\vec{R}$ calculée comme suit:

$$\vec{W} = \frac{\vec{V}}{|\vec{V} \cdot \vec{N}|} \tag{2.12}$$

$$\vec{R} = \vec{W} + 2\vec{N} \tag{2.13}$$

L'intensité transmise $I_t$ est donnée par:

$$I_t = k_t T \tag{2.14}$$

où $k_t$ est le coefficient d'intensité transmise et $T$ est l'intensité provenant de la direction $\vec{P}$:

$$\vec{P} = \frac{\vec{N} + \vec{W}}{\sqrt{k_n^2 ||\vec{W}||^2 - ||\vec{W} + \vec{N}||^2}} - \vec{N} \tag{2.15}$$

La grandeur $k_n$ se calcule comme

$$k_n = \frac{n_2}{n_1} \tag{2.16}$$

où $n_1$ (respectivement $n_2$) est l'indice de réfraction du milieu dans lequel se trouve le rayon de départ (respectivement le rayon réfracté).

### 2.3.6 Remarques

Pour que les objets éloignés n'aient pas une trop grande influence, on multiplie toutes les intensités par un facteur inversément proportionnel à la distance (le long des rayons) de l'observateur.

Toutes les considérations qui ont été faites sont valables pour une représentation de la scène en niveaux de gris (noir et blanc). Pour avoir une représentation en couleurs les équations 2.6, 2.8 et 2.9 deviennent des équations vectorielles, où $I$, $I_a$, $I_d$, $I_s$, $I_r$, $I_t$, $C_s$ et $C_r$ sont des vecteurs à trois composantes: contribution rouge, verte et bleue, chacune de ces valeurs se trouvant dans l'intervalle $[0, 1]$.

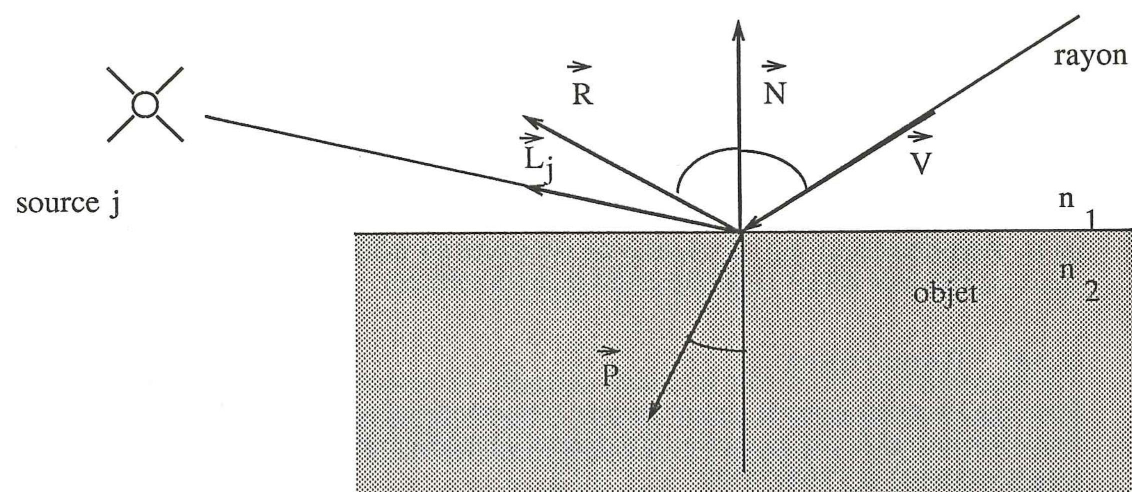


Figure 2.2: vecteurs intervenant dans le modèle d'illumination



# Chapitre 3

# "Ray tracing" de surfaces généralisées

Pour simplifier les notations, on suppose que le rayon $R(t)$ coïncide avec l'axe des "x", c'est-à-dire que l'origine du rayon $R_o$ vaut $(0,0,0)$ et que la direction $V$ vaut $(1,0,0)$, donc $R(t) = (t,0,0)$. Notons que l'on peut toujours se ramener à cette situation en transformant le système de coordonnées par une translation et une rotation.

## 3.1 Définitions

### 3.1.1 Représentation implicite

On dira qu'une surface $S$ est *sous forme implicite*, si elle est donnée par l'ensemble des points $(x,y,z)$ tels que

$$F(x,y,z) = 0 \tag{3.1}$$

où $F$ est une fonction différentiable quelconque à valeurs réelles.

Un cas particulier important est le cas où $F$ est un *polynôme de degré $N$ en $x$, $y$ et $z$*, c'est-à-dire donnée par:

$$F(x,y,z) = \sum_{0 \leq i+j+k \leq N} a_{ijk} x^i y^j z^k \tag{3.2}$$

où $N$ est un entier naturel quelconque et $a_{ijk}$ sont des coefficients réels. On peut citer comme exemples, la sphère: $F(x,y,z) = (x-x_o)^2+(y-y_o)^2+(z-z_o)^2-R^2$, et le plan: $F(x,y,z) = ax + by + cz + d$ avec $a$, $b$, $c$, et $d$ des réels.

### 3.1.2 Représentation paramétrique

On dira qu'une surface $S$ est *sous forme paramétrique*, si elle est donnée par tous les points $(x, y, z)$ tels que:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{pmatrix} \quad (3.3)$$

où $x(u,v)$, $y(u,v)$ et $z(u,v)$ sont des fonctions différentiables quelconques à valeurs réelles, $(u,v)$ prennent leurs valeurs dans un sous-ensemble $\Delta$ du plan (par exemple $\Delta = [0,1] \times [0,1]$) appelé *espace des paramètres*.

Dans la plupart des applications les fonctions $x(u,v)$, $y(u,v)$ et $z(u,v)$ sont des *polynômes bivariés*, c'est-à-dire du type:

$$x(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} x_{ij} u^i v^j \quad (3.4)$$

$$y(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} y_{ij} u^i v^j \quad (3.5)$$

$$z(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} z_{ij} u^i v^j \quad (3.6)$$

où $n$ et $m$ sont des entiers naturels quelconques, et où $x_{ij}$, $y_{ij}$ et $z_{ij}$ sont des coefficients réels. Le cas particulier où $n = m = 3$ est le plus fréquent, on parle alors de *spline bicubique*.

## 3.2 Problème de l'intersection

### 3.2.1 Surfaces sous forme implicite

Pour calculer l'intersection du rayon $R(t)$ avec $S$, on cherche d'abord les points $(x, 0, 0)$ se trouvant sur $S$, c'est-à-dire tels que $f(x) = F(x, 0, 0) = 0$, puis on choisit la solution avec $x > 0$ minimale. Il s'agit donc de trouver la plus petite solution réelle positive de l'équation

$$f(x) = 0 \quad (3.7)$$

Dans le cas où $F$ est polynôme de degré $N$, on peut être encore plus précis en posant comme dans le cas général $y = 0$ et $z = 0$ dans 3.2. On obtient ainsi une fonction polynômiale

$$f(x) = \sum_{i=0}^{N} b_i x^i \quad (3.8)$$

où $b_i = a_{i00}$. Le problème est donc ramené à la recherche de la plus petite racine réelle positive d'un polynôme à coefficients réels.



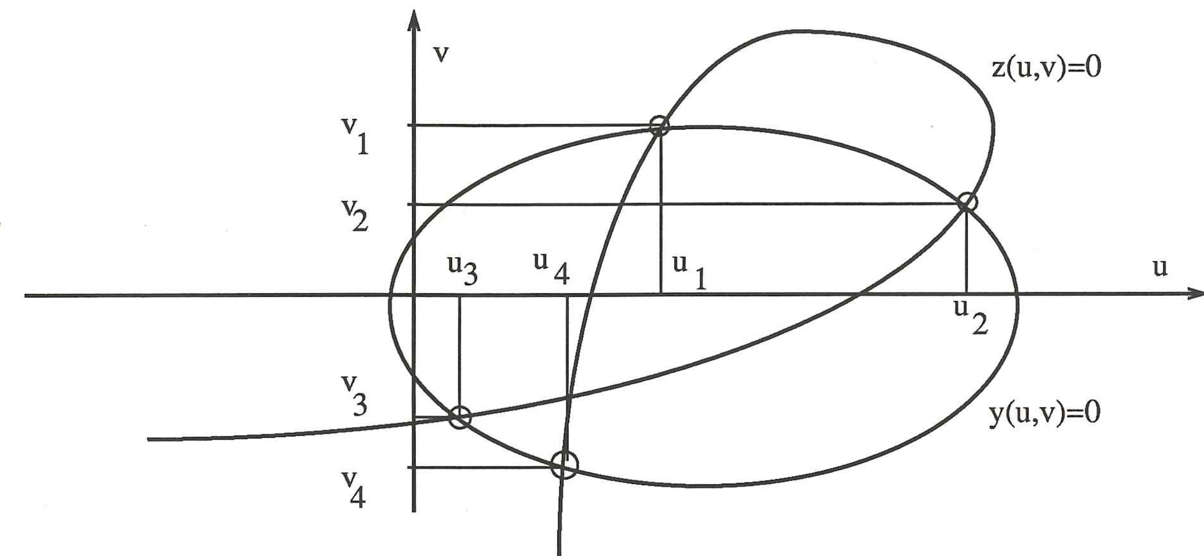


Figure 3.1: Intersections de deux courbes implicites

### 3.2.2 Surfaces sous forme paramétrique

Comme dans le cas de la représentation implicite, on cherche l'intersection parmi les points $(x, 0, 0)$ sur la surface $S$. Ce qui est équivalent à chercher les valeurs de $(u, v)$ dans $\Delta$ tels que les équations suivantes soient satisfaites:

$$x(u,v) = t \quad (3.9)$$

$$y(u,v) = 0 \quad (3.10)$$

$$z(u,v) = 0 \quad (3.11)$$

On voit en fait que l'on détermine ces valeurs $(u, v)$ avec les deux équations 3.10 et 3.11, et que l'on sélectionne la solution avec t minimal à l'aide de l'équation 3.9. Tout le travail réside donc dans la recherche des zéros d'une fonction $G(u,v) = (y(u,v), z(u,v))$, c'est-à-dire à trouver les solutions de

$$G(u,v) = 0 \quad (3.12)$$

Les solutions de l'équation 3.12 peuvent s'interpréter géométriquement comme l'ensemble des points d'intersections de deux courbes implicites $y(u,v) = 0$ et $z(u,v) = 0$ (c.f. figure 3.1).

## 3.3 Méthodes itératives

Pour résoudre des équations du type 3.7 ou 3.12 numériquement on utilise souvent des méthodes itératives.

### 3.3.1 La méthode de Newton

Considérons d'une manière plus générale une fonction

$$H : D \subset \mathbf{R}^p \longrightarrow \mathbf{R}^p \tag{3.13}$$

avec $p$ un entier naturel. Le problème est de trouver les points $\xi^*$ dans $D$ tels que

$$H(\xi^*) = 0 \tag{3.14}$$

On suppose que l'on connaisse une approximation $\xi^0$ d'un des $\xi^*$, la *méthode de Newton* est une suite d'approximations

$$\xi^{k+1} = \xi^k - A^{-1}H(\xi^k), \qquad k = 0, 1, 2, \ldots \tag{3.15}$$

où $A$ est une matrice $p \times p$ à coefficients réels, souvent A vaut $H'(\xi^0)$, c'est-à-dire la matrice Jacobienne de $H$ en $\xi^0$, d'autres choix possibles de $A$ sont décrits en détails dans l'ouvrage [8].

### 3.3.2 Choix de la valeur initiale

Le grand problème de cette méthode est le choix de la valeur initiale $\xi^0$ assurant une convergence de la suite des $\xi^k$ vers la solution $\xi^*$. Toth dans [14] a résolu ce problème pour les surfaces paramétriques en subdivisant l'espace des paramètres $\Delta$ jusqu'à ce qu'un certain critère soit satisfait (test de *Krawczyk-Moore*). Sweeney et Bartels dans [12] ont une approche analogue, mais leur méthode ne s'applique qu'aux surfaces "B-splines".

## 3.4 Racines communes de deux polynômes bivariés

On a vu que dans le cas de la représentation paramétrique où $x(u,v)$, $y(u,v)$ et $z(u,v)$ sont des polynômes, on devait rechercher les racines communes de deux polynômes en $(u,v)$. Kajiya dans [6] a développé un algorithme reposant sur le concept de *résultantes de polynômes*. La résultante des deux polynômes donne un polynôme univarié en $u$, ce polynôme permet de calculer les composantes "$u$" des racines communes. Une fois ces composantes déterminées on les substitue dans les polynômes $y(u,v)$ et $z(u,v)$ de départ afin d'obtenir deux polynômes univariés en $v$, Kajiya calcule ensuite le PGCD de ces deux polynômes, obtenant ainsi les composantes "$v$" correspondants aux $u$ déjà calculés.



En annexe de ce travail on peut trouver une autre approche qui utilise le concept de *base de Groebner* associée aux polynômes $y(u,v)$ et $z(u,v)$.

On voit que ces deux algorithmes nécessitent une bonne procédure pour rechercher les racines réelles de polynômes univariés.

## 3.5 Racines réelles de polynômes

### 3.5.1 Les suites de Sturm

Les suites de Sturm sont à la base des algorithmes de recherche de racines réelles de polynômes.

La *suite de Sturm* associée au polynôme

$$P(x) = a_0x^p + a_1x^{p-1} + \cdots + a_p \tag{3.16}$$

est une suite

$$V_0(x), V_1(x), \ldots, V_q(x) \tag{3.17}$$

de polynômes avec $q \leq p$. Les deux premiers éléments de la suite sont donnés par:

$$V_0(x) = P(x) \tag{3.18}$$

$$V_1(x) = P'(x) \tag{3.19}$$

alors que les polynômes restants sont calculés à l'aide des équations:

$$V_{k-2}(x) = Q_{k-1}(x)V_{k-1}(x) - V_k(x), \qquad k = 2, \ldots, q \tag{3.20}$$

Cette dernière expression n'est rien d'autre que l'algorithme d'Euclide (au signe près) pour déterminer le PGCD du polynôme $P(x)$ et de sa dérivée $P'(x)$.

On note par $\sigma(x_0)$ le nombre de changements de signes dans la suite

$$V_0(x_0), V_1(x_0), \ldots, V_q(x_0) \tag{3.21}$$

où l'on a omis les éléments nuls. Les suites de Sturm permettent de localiser les racines réelles d'un polynôme à cause du résultat suivant:

$$\sigma(a) - \sigma(b) = \text{ nombre de racines réelles dans } [a,b] \tag{3.22}$$

Pour une preuve de ce dernier résultat et des explications plus détaillées on réfère le lecteur à l'ouvrage [15].

### 3.5.2 Isolement des racines

Pour calculer toutes les racines réelles d'un polynôme on calcule d'abord un ensemble

$$I = \{]a_i, b_i]\}_{i=1}^q \tag{3.23}$$

d'intervalles, chacun contenant exactement une racine réelle $r_i$ du polynôme $P(x)$, grâce à la suite de Sturm:

**Algorithme d'isolement des racines**

$I := \emptyset$
$]-c, c]$ contient toutes les racines réelles de $P(x)$
$J := \{]-c, c]\}$
tant qu'il existe un intervalle $]a, b]$ dans $J$ faire
  sortir $]a, b]$ de $J$
  $m := \frac{a+b}{2}$
  $r_1 := \sigma(a) - \sigma(m)$
  $r_2 := \sigma(m) - \sigma(b)$
  si $r_1 = 1$ alors mettre $]a, m]$ dans $J$
  si $r_1 > 1$ alors mettre $]a, m]$ dans $I$
  si $r_2 = 1$ alors mettre $]m, b]$ dans $J$
  si $r_2 > 1$ alors mettre $]m, b]$ dans $I$
fait

Il est clair que dans le cas où l'on ne cherche que la plus petite racine positive, on ne calcule pas tous les intervalles mais seulement l'intervalle $]0, d]$ contenant une seule racine.

Notons tout de même que l'emploi des suites de Sturm n'est pas absolument nécessaire, Hanrahan dans [5], par exemple, utilise l'algorithme modifié d'Uspensky par Collins, on peut trouver encore d'autres méthodes dans la référence [2].

### 3.5.3 Recherche de la racine

Si l'ensemble $I$ est vide, $P(x)$ n'a aucune racine réelle, sinon on recherche les racines contenus dans les intervalles de $I$. On donne ici la méthode la plus simple pour rechercher la racine $r$ contenue dans un intervalle $]a, b]$ de $I$ avec une précision donnée $\varepsilon$: *méthode de la bissectrice*.

tant que $|a - b| > \varepsilon$ faire
  $m := \frac{a+b}{2}$
  si $P(m)P(b) < 0$ alors $a := m$
  sinon $b := m$
fait

à la fin de cet algorithme $a$ (ou $b$) est une approximation de la racine recherchée.

On peut encore citer comme autres méthodes: la methode de *regula-falsi* (c.f. [3]) ou *l'itération de Laguerre* utilisée par Kajiya dans [6].



## 3.6 Problème du vecteur normal

### 3.6.1 Représentation implicite

Le calcul du vecteur normal en un point $(x_o, y_o, z_o)$ de la surface $S$ est donné par le vecteur gradient en ce point de $F$:

$$\vec{N} = (\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z})\,|_{(x_o, y_o, z_o)} \tag{3.24}$$

Le calcul du vecteur normal peut donc poser un problème, si $(x_0, y_0, z_0)$ est un point critique de $F$, c'est-à-dire si $\vec{N} = \vec{0}$. Ceci peut même être le cas lorsque $F$ est un polynôme. On note que lorsque $F$ possède des points critiques, $S$ n'est pas une surface au sens mathématique du terme.

### 3.6.2 Représentation paramétrique

Le vecteur normal en un point $(x_o, y_o, z_o)$ de la surface $S$ donné par la valeur de $(u_o, v_o)$ est le vecteur normal au plan tangent de la surface au point $(x_o, y_o, z_o)$ (c.f. figure 3.2):

$$\vec{N} = (\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}) \wedge (\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v})\,|_{(u_o, v_o)} \tag{3.25}$$

Comme pour le cas implicite, on a des problèmes lorsque le plan tangent est dégénéré, c'est-à-dire si les vecteurs de l'équation 3.25 sont colinéaires ou nuls donc $\vec{N} = \vec{0}$.

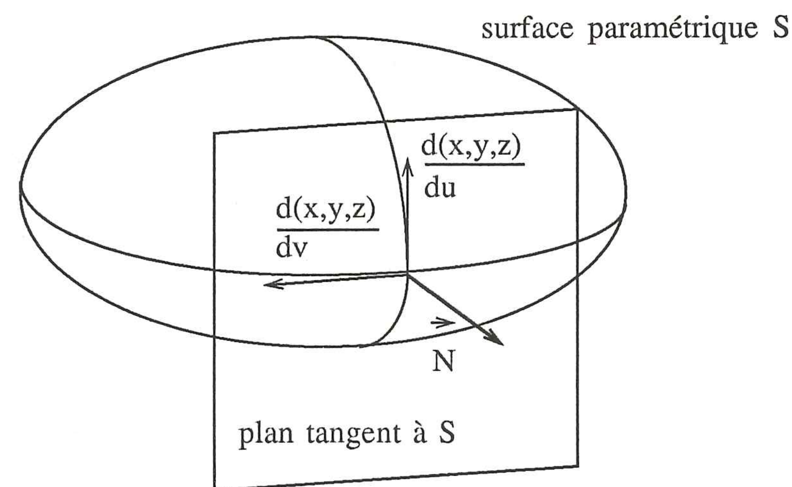


Figure 3.2: Vecteur normal



# Chapitre 4

# Implantation

Dans ce chapitre on décrit le logiciel de "ray tracing" permettant d'implanter les divers algorithmes exposés dans le chapitre précédent. Cette implantation s'est initialement inspirée du mémoire de licence de Brunet ([1]).

## 4.1 Description du logiciel

### 4.1.1 Description globale

La figure 4.1 montre les liens entre les différents modules du logiciel. Le module **`InitScene`** lit le fichier de commande donné par l'utilisateur (c.f. prochaine section) et initialise les structures de données en conséquence. **`InitScreen`** alloue les ressources nécessaires à afficher une image sur l'écran, elle dépend par conséquent du matériel utilisé; **`CleanUp`** désalloue ces ressources et termine le logiciel.

Le module **`TraceScene`** s'occupe du traçage à proprement parler de la scène. Les paramètres de la caméra sont tout d'abord transformés à l'aide du module **`ProjectPlane`**, puis pour chaque "pixel" de la fenêtre on trace un rayon à l'aide de **`TraceRay`**, **`Plot`** affiche/sauve la couleur retournée par le rayon.

**`TraceRay`** recherche tout d'abord l'intersection (s'il y en a une) la plus proche d'un objet de la scène avec le rayon donné (**`GetIntersection`**). L'intersection et le vecteur normal sont fournis par des modules propres à l'objet intercepté. L'intensité au point d'intersection est déterminé à l'aide du module **`Intensity`**.

### 4.1.2 Code propre à un objet

Pour ajouter un nouvel objet au logiciel il faut fournir d'abord deux modules: un qui permet de calculer l'intersection d'un rayon quelconque avec l'objet, et un autre qui calcule le vecteur normal en un point donné de la surface de l'objet. Le module d'intersection doit être dans le format suivant (en langage C):

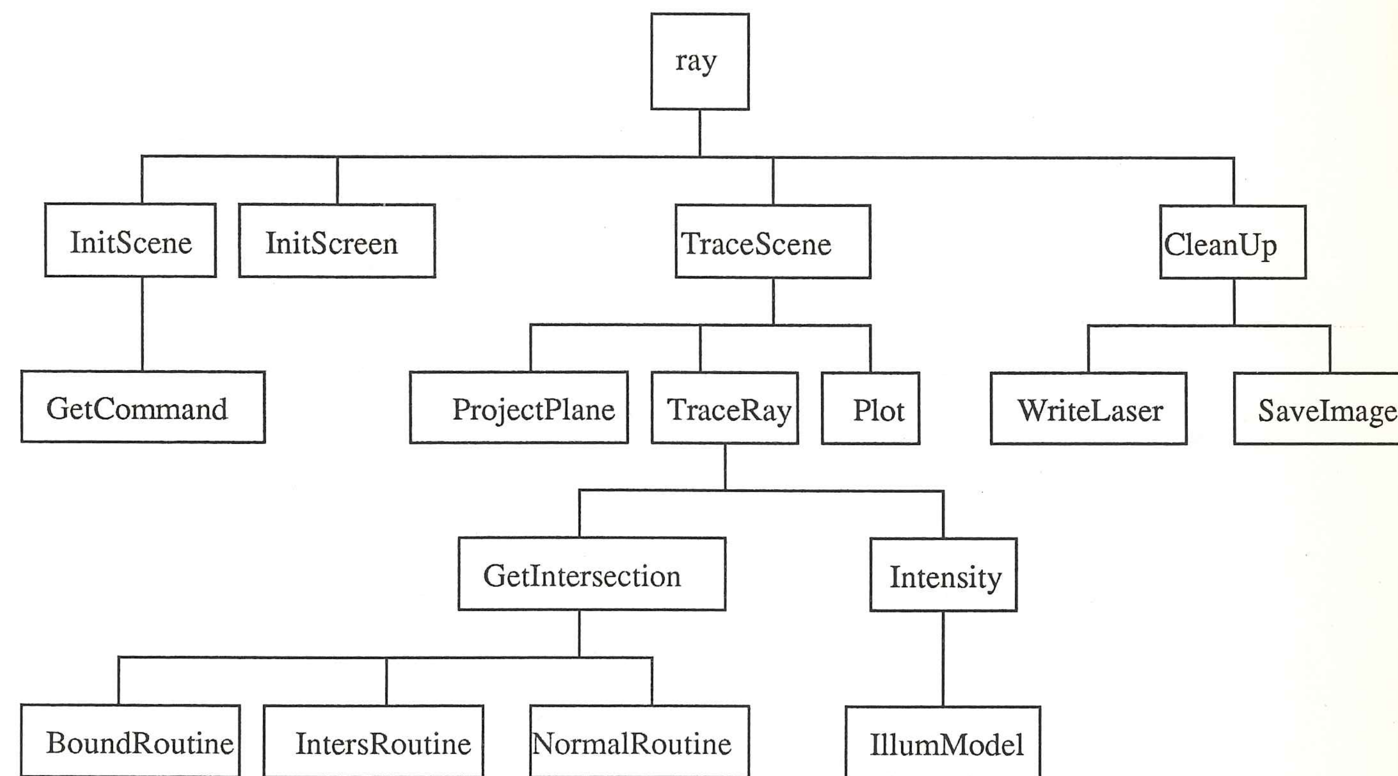


Figure 4.1: Structure générale du logiciel



`BOOLEAN IntersRoutine ( ray, param, patch, NbObj, shadow, t1, t2 )`

`RAY *ray;` *structure définissant le rayon*

`void *param;` *structure des paramètres propres à l'objet*

`PATCH *patch;` *structure retournée par le module contenant le point d'intersection*

`short NbObj;` *index de l'objet dans la table des objets de la scène*

`BOOLEAN shadow;` *indique si l'on calcule les ombres*

`double t1, t2;` *on ne retient que les points d'intersections avec* $t \in$ [`t1`,`t2`]

le module retourne **FALSE** si aucune intersection n'a eu lieu, sinon il retourne **TRUE**. Le module du vecteur normal quant à lui a le format:

`void NormalRoutine ( param, patch )`

`void *param;` *structure des paramètres propres à l'objet*

`PATCH *patch;` *le module rajoute le vecteur normal calculé à cette structure*

En plus de ces deux routines il faut encore fournir un module qui lit les paramètres propres à l'objet du fichier de commandes (c.f. prochaine section). Pour plus de détails voir le code source donné en annexe.

## 4.2 Interface avec l'utilisateur

L'utilisateur communique avec le logiciel à travers un fichier de commandes. Chaque commande a le format suivant:

*#nom de la commande*

*paramètres de la commande*

### 4.2.1 Spécification de la scène

Avec la commande **world** on donne la couleur du fond et l'on spécifie si l'on désire des ombres ou pas (le calcul des ombres augmente le temps calcul). Son format exact est:

```
#world
BackGnd = (R,G,B)
Shadows = ON | OFF
```

La commande **camera** permet de donner la position de l'observateur et les caractéristiques de la caméra:

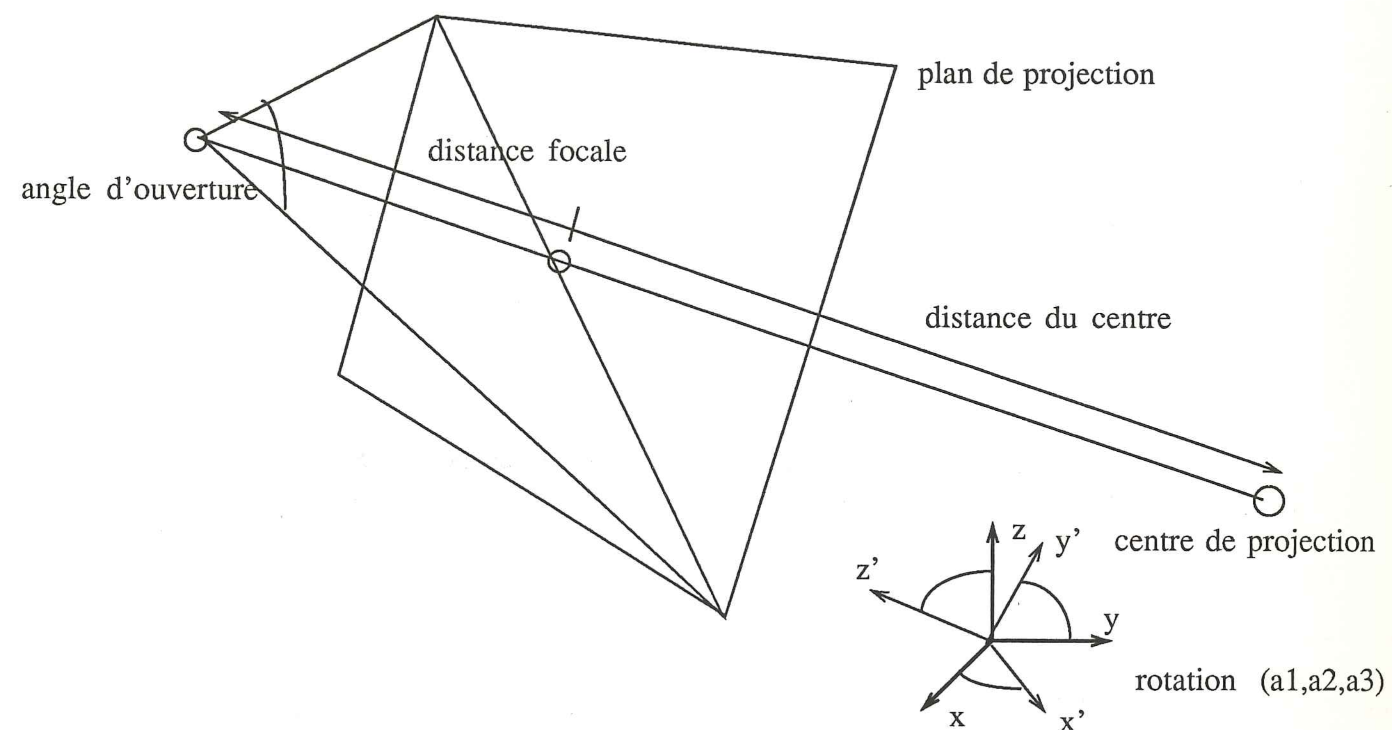


Figure 4.2: paramètres de la caméra

```
#camera
Focal = distance focale
Distance = distance du centre
ViewField = angle d'ouverture
Rotation = (a1,a2,a3)
Center = (x,y,z)
```

La figure 4.2 donne la signification de chaque paramètre.

Les sources lumineuses sont spécifiées à l'aide de la commande `light`:

```
#light
Source = (x,y,z)
Color = (R,G,B)
```

Pour décrire un objet de la scène on utilise la commande `object`. Cette commande a deux parties: dans la première on spécifie le type d'objet et les



paramètres propres à l'objet, alors que dans la deuxième partie on donne les paramètres décrivant le modèle d'illumination de l'objet. Son format est:

```
#object
Type = type de l'objet
paramètres propres à l'objet
Illumination = type d'illumination
Cs = (R,G,B)
Cr = (R,G,B)
Ka = k_a Kd = k_d Ks = k_s Kt = k_t
n = n Kn = k_n
Bounding volume = NONE | BOX
[x_min,x_max]x[y_min,y_max]x[z_min,z_max]
```

Les types d'objets possibles sont: **SPHERE**, **PLANE**, **POLYGON**, **ALGEBRAIC_SURFACE** et **BEZIER_PATCH**. Voyons pour chacun d'eux les paramètres propres. Pour la sphère (**SPHERE**) on a:

```
Center = (x_o,y_o,z_o)
Radius^2 = R^2
```

Pour le plan (**PLANE**):

```
Origin = (O_x,O_y,O_z)
Normal = (N_x,N_y,N_z)
```

Pour le polygone à quatres côtés $a$, $b$, $c$ et $d$ (**POLYGON**):

```
a = (a_x,a_y,a_z)
b = (b_x,b_y,b_z)
c = (c_x,c_y,c_z)
d = (d_x,d_y,d_z)
```

la surface sous forme implicite (**ALGEBRAIC_SURFACE**) n'a pas de paramètres propres, la fonction polynomiale $F$ est directement codée dans le programme (et ne peut donc être modifiée par le fichier de commandes). Le "patch" de Bézier (**BEZIER_PATCH**) a un seul paramètre:

```
Control points in nom de fichier
```

Le fichier contient 16 points de controle, un point par ligne.

Les types d'illumination sont BLACKBODY (objet noir), AMBIANT ($I = I_a$), DIFFUSE ($I = I_a + I_d$), SPECULAR ($I = I_a + I_d + I_s$), MIRROR ($k_t = 0$) et TRANSPARENT. Les paramètres d'illumination ont exactement la signification qu'on leur a donnée dans le chapitre 2.

Le dernier paramètre de la commande `object` donne l'objet enveloppant. Dans l'implantation courante deux choix seulement sont possibles: pas d'objet enveloppant (NONE et la dernière ligne de la commande est omise) ou l'objet est une boîte (BOX). L'objet enveloppant est un objet plus simple qui contient l'objet en question. Si un rayon ne touche pas l'objet enveloppant, on sait automatiquement qu'il n'intercepte pas l'objet enveloppé.

### 4.2.2 Affichage de la scène

La scène est toujours affichée en noir et blanc sur l'écran dans une fenètre spécifiée par la commande `viewport`:

```
#viewport
Size = (SizeX,SizeY)
Origin = (OrgX,OrgY)
Resolution = (ResX,ResY)
```

Pour avoir une idée grossière de la scène, on met $ResX$ et $ResY$ à des valeurs différentes de 1, c'est-à-dire que l'intensité retournée par chaque rayon couvrira $ResX \times ResY$ pixels de la fenètre.

L'image peut être sauvée à l'aide de la commande `save`:

```
#save
File Name = nom de fichier
```

Chaque "pixel" de la fenètre est sauvé comme une suite de 3 "bytes" (composantes R, G et B) dans le fichier donné, ceci est utile pour effectuer des traitements ultérieurs à l'image. La commande `dump` permet d'avoir un fichier "post-script" pour une impression sur l'imprimante laser:

```
#dump
File Name = nom de fichier
```

## 4.3 Résultats

Toutes les images ont été générées sur un SUN4. La résolution de toutes les images est de 840 x 840 pixels. La figure 4.3 montre une image typique de "ray tracing", avec un plan, deux sphères et deux polygones. Il a fallu environ une demi-heure pour la générer. La possibilité de mettre une texture sur le plan de

Figure 4.3: Image typique de "ray tracing"

base n'a été ajoutée qu'au tout dernier moment. Pour l'utiliser il faut que le plan soit parallèle au plan $z = 0$, de plus il faut specifier une illumination de type TEXTURE et il faut donner un fichier contenant la texture avec la commande `texture`:

```
#texture
Size = TailleX x TailleY
File Name = nom de fichier
Unit = taille de l'image sur le plan
```

Les figures 4.5 et 4.6 montrent des images d'une surface sous forme implicite: le tore centré à l'origine de diamètre 1 et de grand rayon 2. La génération de l'image de la figure 4.5 a pris 3 heures, alors que celle de la figure 4.6 a pris 5 heures. Pour trouver la représentation implicite, on cherche une fonction $g(x, y)$ à valeurs réelles qui est positive sur la couronne hachurée de la figure 4.4 et qui est négative ailleurs, puis la fonction $F$ du tore est donnée par:

$$F(x, y, z) = g(x, y) - z^2 \quad (4.1)$$

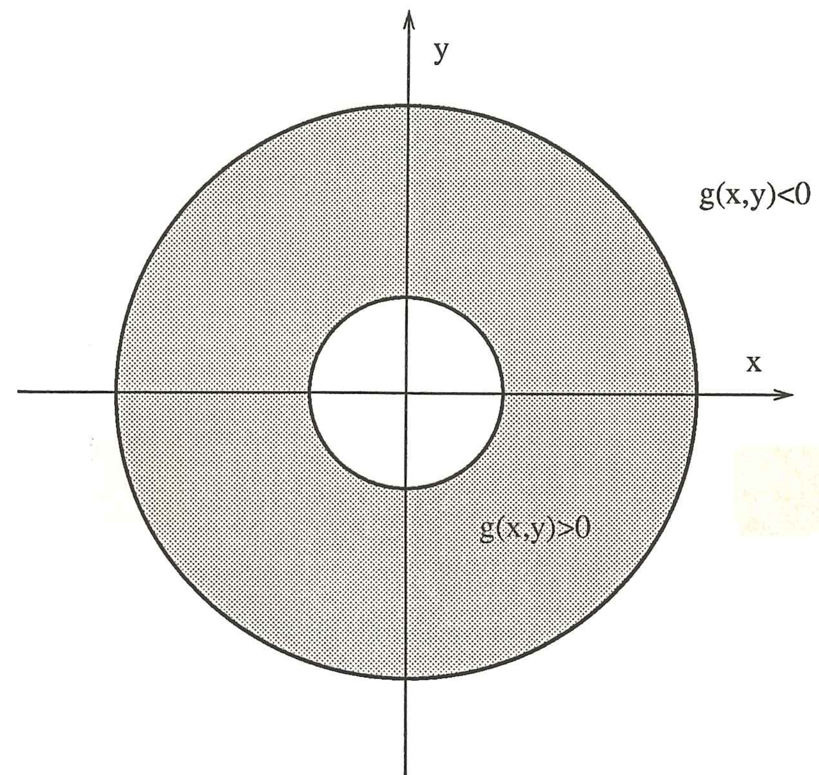


Figure 4.4: Calcul de la forme implicite du tore

La solution pour $g(x, y)$ est simplement l'équation du cercle de rayon 3 multiplié par l'équation du cercle de rayon 1:

$$g(x, y) = -(x^2 + y^2 - 1)(x^2 + y^2 - 9) \tag{4.2}$$

Donc pour obtenir $F$ il suffit de substituer 4.2 dans l'équation 4.1.

Pour des résultats concernant les surfaces sous forme paramétrique, on réfère le lecteur à l'article donné en annexe.



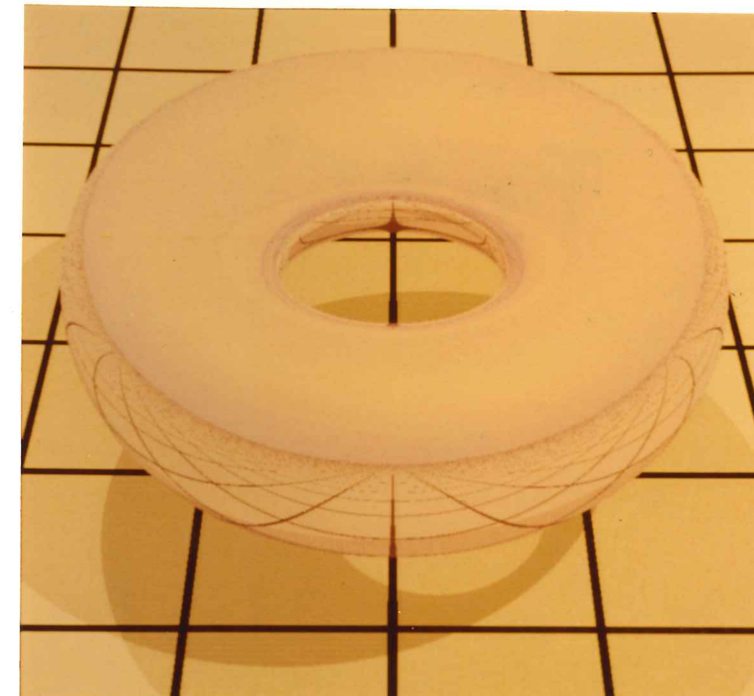

Figure 4.5: le tore réflectif

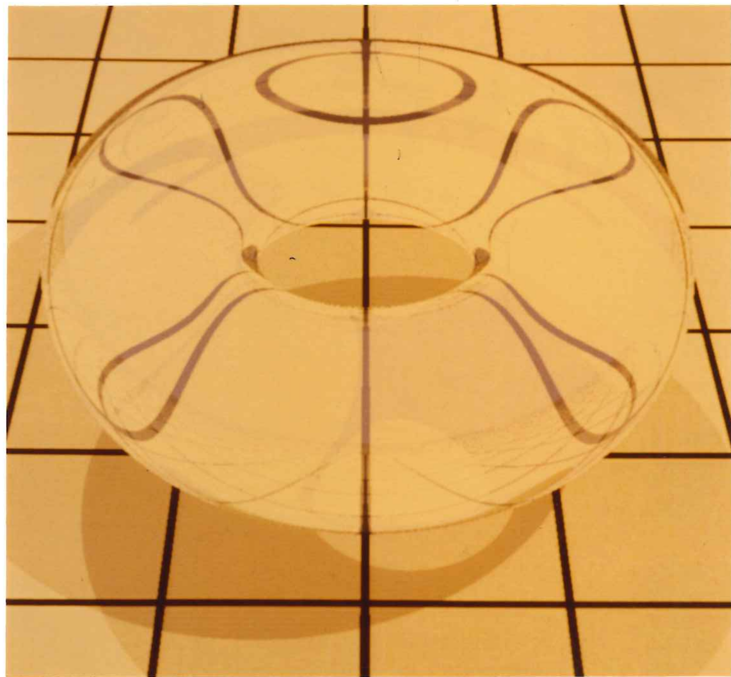

Figure 4.6: le tore transparent



# Chapitre 5

# Conclusion

Dans le présent travail on a montré comment on pouvait rendre par la méthode de "ray tracing" des surfaces données par deux types de représentations: implicite et paramétrique. Néanmoins les procédures données ne tiennent pas comptes des instabilités numériques provenant de l'utilisation des nombres à virgule flottante.

Dans l'implantation on n'a considéré que les surfaces oú les fonctions sont polynomiales, et nous avons vu que dans les deux cas (implicite et paramétrique) on était amené à calculer les racines réelles d'un polynôme. La méthode utilisée (les suites de Sturm) n'est stable (dans notre implantation) que lorsque le degré du polynôme est inférieur à 9.

Ceci ne pose donc pas de problèmes pour les représentations implicites où $F$ est un polynôme de degré inférieur à 9. Ceci est notamment le cas du tore, qui est une surface de degré 4.

Dans le cas des surfaces sous forme paramétriques par contre on a beaucoup plus de problèmes car le degré du polynôme résultant est beaucoup plus élevé que celui de la surface. Kajiya dans [6] prouve que dans le cas où $n = m = 3$ le polynôme résultant est de degré 18 dans le cas général. Ce qui provoque évidemment des instabilités numériques. On a quand même obtenu de bons résultats pour des cas particuliers (c.f. article donné en annexe).

La représentation paramétrique, bien que très populaire dans les visualisations "fil de fer" (wire frame), ne semble pas adéquat pour le "ray tracing".

Il n'est donc pas étonnant que les méthodes les plus performantes pour rendre des surfaces sous la forme paramétriques par le "ray tracing" modifient dans un premier temps la représentation paramétrique en une autre. Snyder et Barr dans [11] triangularisent d'abord les surfaces, et raménent ainsi le problème au traçage d'un grand nombre de polygones (triangles).

On peut également rendre la surface paramétrique implicite. Sederberg et Anderson dans [10] ont adopté cette approche pour un cas particulier: les "patches" de Steiner.

Ainsi bien que la solution théorique est assez directe, l'implantation pratique pose un certain nombre de problèmes, qui sont surtout dûs aux instabilités numériques. Il semble que la modélisation (représentation de la surface) et le rendu ("ray tracing") doivent être considérées en même temps, afin d'obtenir l'algorithme le plus efficace possible.



# Annexe A

# Article

# Ray tracing polynomial patches using Groebner bases

J. Stam

October 18, 1988

## 1 Introduction

Ray tracing has become in this past decade one of the most popular methods to render highly realistic images. An important drawback of this method, however, is the tremendous amount of computing that is necessary, even to render a simple image. This is why most of the ray-traced images contain only simple objects, such as spheres, planes or cones. Recently, however, many efforts have been made to increase the amount of "ray-traceable" objects. The class of objects generated by "spline" surfaces is particulary interesting, because of the variety of different objects it contains.

Several approaches have been studied and implemented to ray trace spline surfaces. These include the works of Whitted [11], Toth [10], Sweeney and Bartels [9], Kajiya [6], Snyder and Barr [8] and of course many other papers not mentioned here.

In this paper we will consider Kajiya's approach, which reduces the intersection problem to the search of the common roots of two algebraic curves. This approach has the advantage of being very general (no restriction on the degree or on the type of spline surface) and very direct with only a few special cases to consider. However the solution proposed by Kajiya included the resolution of a polynomial equation of a high degree, resulting in long computations and numerical instabilities. In this paper we will try to overcome the high degree problem by reducing the two algebraic equations by a simplification algorithm of computational algebra. Furthermore by doing so we get a criterium to determine whether the ray misses the surface.

## 2 Ray/surface intersection problem

Before stating the intersection problem we suppose that we are given a polynomial patch

$$S(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{ij}u^i v^j$$

where $P_{ij} = (P_{ij}^x, P_{ij}^y, P_{ij}^z)$ and a ray in its usual form

$$R(t) = R_0 + Vt$$

where $R_0 = (R_0^x, R_0^y, R_0^z)$ and $V = (V^x, V^y, V^z)$. The problem is to find the nearest intersection of the ray with the given surface — that is, closest to the origin $R_0$ of the ray. This can be expressed by the following equation:

$$S(u,v) - R(t) = 0 \quad (1)$$

which can be broken into its component parts

$$\sum_{i=0}^{n}\sum_{j=0}^{m} P_{ij}^x u^i v^j - R_0^x - V^x t = 0 \quad (2)$$

$$\sum_{i=0}^{n}\sum_{j=0}^{m} P_{ij}^y u^i v^j - R_0^y - V^y t = 0 \quad (3)$$

$$\sum_{i=0}^{n}\sum_{j=0}^{m} P_{ij}^z u^i v^j - R_0^z - V^z t = 0 \quad (4)$$

By solving this nonlinear system we get a set of solutions $(u, v, t)$, from which we choose the one that has the smallest positive t. Let us suppose that $V^x \neq 0$, so that we can simplify the above system by multiplying 3 (resp. 4) with $V^x$ and then substracting 2 multiplied by $V^y$ (resp. by $V^z$). With these manipulations we eliminate the parameter t and reduce to the two following equations:

$$F_1(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_{ij}u^i v^j = 0 \quad (5)$$

$$F_2(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} b_{ij}u^i v^j = 0$$

The problem is thus reduced to the search of the common roots of two algebraic implicit curves. Once we have determined these roots $(u, v)$ we can calculate the corresponding t with equation 2 for example.



In the rest of this paper we will describe the algorithm which solves equation 5. In order to have an efficient algorithm, we want it to fulfill the following two conditions:

*C1* There must be a criterium which tells us that 5 has no solution at all (i.e. the ray misses the surface).

*C2* If *C1* is satisfied then we must have an algorithm which guarantees us to find all the solutions of 5.

## 3 Definitions and theoretical results

Here we present the theoretical background for the algorithm described in the next section.

The $F_1$ and $F_2$ in equation 5 can be viewed as elements of $K[u, v]$ the ring of bivariate polynomials with coefficients in the field $K$ (in our case $K$ will be the field of the rationals). For the sake of brevity each element of $K[u, v]$ will simply be called a *polynomial*. An important subset of $K[u, v]$ is

$$I = \{pF_1 + qF_2 \mid p, q \in K[u, v]\} = (F_1, F_2)$$

which is called the *ideal generated* by $F_1$ and $F_2$. We note that each polynomial in $I$ vanishes in the common roots of $F_1$ and $F_2$.

For the definitions which will follow it is necessary to define an ordering $<_M$ of the monomials. This ordering must be *acceptable* in the sense that $f <_M g \implies hf <_M hg$ for all monomials $f$, $g$ and $h$. We can for example consider the *Total Lexicographical Order (TLO)*:

$$1 < u < u^2 < u^3 < \cdots < v < uv < u^2v < \cdots < v^2 < uv^2 < u^2v^2 < \cdots$$

**Definitions:**

1. $LM(p)$ : *leading monomial* of $p \in K[u, v]$, relatively to $<_M$. As the coefficients are in a field $K$ we are allowed to suppose that the coefficient of $LM(p)$ is equal to 1
2. $Deg_u(p)$ and $Deg_v(p)$ : power of $u$ (resp. $v$) in the monomial $LM(p)$.
3. $F = \{F_1, F_2\}$ : *basis of the ideal I.*
4. $p \rightarrow_F q$ : *p reduces to q modulo the basis F* where $p, q \in K[u, v]$, the reduction process $\rightarrow_F$ is defined as follows:
   - if a monomial of $p$ is a multiple of a $LM(F_i)$, i.e. equal to $\lambda_i LM(F_i)$, then replace $p$ by $p - \lambda_i F_i$.
   - Repeat the above step until no such $\lambda_i$ exists.

- At the end we obtain the polynomial $q$.

5. $NF(p, F)$ : *normal form of p* with respect to the basis F, i.e. $p \rightarrow_F NF(p, F)$

6. $SP(F_i, F_j)$ : *S-polynomial* of $F_i$ and $F_j$, calculated as follows
   if $LM(F_i) = u^{i_1}v^{i_2}$ and $LM(F_j) = u^{j_1}v^{j_2}$ then

$$SP(F_i, F_j) = u^{d_1-i_1}v^{d_2-i_2}F_i - u^{d_1-j_1}v^{d_2-j_2}F_j$$

   where $d_k = \max(i_k, j_k)$, $k = 1, 2$.

7. A basis G is said to be a *Groebner basis* (with respect to ideal $I$) if the following holds:

$$p \in I \implies p \rightarrow_G 0$$

   i.e if all polynomials in the ideal $I$ reduce to the zero-polynomial (zero for all $u$ and $v$). The reciprocal form $p \rightarrow_G 0 \implies p \in I$ is true for any basis.

**Example** (with TLO ordering of the monomials)
$F = \{F_1, F_2\}$, where
$F_1 = uv + uv^2 - u^3$ and $F_2 = -u^2 + 1 + u^3v$
$LM(F_1) = uv^2$, $LM(F_2) = u^3v$
$Deg_u(F_1) = 1$, $Deg_v(F_1) = 2$, $Deg_u(F_2) = 3$ and $Deg_v(F_2) = 1$.
$SP(F_1, F_2) = u^2F_1 - vF_2 = u^3v + u^2v - v - u^5$
if we put $p = u^2v^2 + uv + 1$ then we can reduce $p$ with $F_1$:

$$q = p - uF_1 = -u^2v + uv + u^4 + 1$$

q can still be reduced by $F_2$, so we continue

$$q + uF_2 = uv + u^4 - u^3 + u + 1 = NP(p, F)$$

this last expression cannot be reduced any further, it is thus the normal form of $p$ with respect to $F$.

We can now consider the following result:

**Theorem 1** *If G is a Groebner basis then we have: 5 has no solutions iff* $1 \in G$

*proof:* It is clear that 5 has no solutions at all iff $1 \in I$ (i.e. $1 = pF_1 + qF_2$ and thus $F_1$ and $F_2$ have no roots in common). By definition of the Groebner basis we have:

$$1 \rightarrow_G 0$$

This reduction occurs iff 1 is in the Groebner basis, i.e. iff $1 \in G$.
We thus have that the condition *C1* stated in the previous section is satisfied.



The problem is now to transform $F = \{F_1, F_2\}$ into a Groebner basis $G$, so that we can apply theorem 1. Buchberger ([1], [2] and [3]) has solved this problem by providing the following algorithm :

**Algorithm of Groebner-Buchberger (GB0)**

*Input:* $F$ any basis of an ideal $I$.
*Output:* $G$ Groebner basis obtained from $F$

*Algorithm:*

$G := F$;
$C := \{(i, j) \mid 1 \leq i < j \leq |G|\}$;

while there exists $(I, J) \in C$ do

  $p := SP(G_I, G_J)$;
  $p := NF(p, G)$;

  if $p \neq 0$ then
    $G := G \cup \{p\}$;
    $C := C \cup \{(i, |G|) \mid 1 \leq i < |G|\}$;
  end if

  $C := C - \{(I, J)\}$;

end while

For the proof that this algorithm effectively computes a Groebner basis see [1]. The set $C$ is called the *set of critical pairs*, $|G|$ denotes the number of elements in $G$.
Although this algorithm terminates in a finite number of steps [1], it is however in practice quite complex. As it is a very general algorithm we shall make some modifications to make it more efficient for our special case ($F$ contains only two polynomials).
In order to calculate all the common roots of 5 we have the following result ([3]):

**Theorem 2** *If G is a Groebner basis then we have: 5 has a finite number of solutions iff there exists an i such that* $LM(G_i) = u^I$ *and a j such that* $LM(G_j) = v^J$.

*proof:* see [3]
a direct corollary is

**Theorem 3** *If $G$ is a Groebner basis and the TLO ordering of the monomials is used then 5 has a finite number of solutions iff $G$ contains a polynomial $G_i \in K[u]$, i.e. a univariate polynomial in $u$.*

Thus by finding the roots of the $G_i$ in theorem 3 we get the $u$ coordinates of the common roots of 5. Then by substituting each $u$ found in one of the $G_j \in F$ containing a $v$ we get a univariate polynomial in $v$, which can be solved to get the corresponding $v$ coordinates. We have thus that condition $C2$ is also satisfied.

# 4 Intersection algorithm

In this section we describe the actual algorithm for finding the intersection with a given surface patch. When a scene contains many patches it is of course very inefficient to apply directly this algorithm to each one. We suppose thus that some patches have been selected by their bounding volumes, we can even suppose that a partial subdivision already has been carried out on the patches (in this last case our algorithm will be applied to these subpatches).

## 4.1 Computing the coefficients

It is possible that the surface is not given in the form $\sum_{i=0}^{n}\sum_{j=0}^{m} P_{ij}u^i v^j$, in this case we have to calculate the $P_{ij}$ coefficients. This is the case for example when the surface is given by a set of control points (Bezier patches, B-splines,...). These calculations must be carried out only once per frame. For each ray however we have to calculate the coefficients $a_{ij}$ and $b_{ij}$ of the two implicit curves $F_1(u,v)$ and $F_2(u,v)$ as described in section 2.

## 4.2 Computing the Groebner basis

First let us consider another reducing procedure: $R2(p,q)$ which reduces polynomials $p$ and $q$ mutually. It is defined as follows:

while $LM(p) = \lambda LM(q)$ do
    $p := p - \lambda q$;
    if $LM(p) < LM(q)$ then exchange $p$ and $q$;
end while

**Example**
$p = u^2v - v + u$ and $q = uv + u^2 + 2$
$p := p - uq = -v - u^3 - u$.
As $LM(p) < LM(q)$ we have to exchange $p$ and $q$:
$p = uv + u^2 + 2$ and $q = -v - u^3 - u$.
$p := p + uq = -u^4 + 2$



Again we have to exchange $p$ and $q$:
$p = -v - u^3 - u$ and $q = -u^4 + 2$
$LM(p)$ is no more a multiple of $LM(q)$ so $R2(p,q)$ terminates.

We can see from this example that $R2(p,q)$ has the effect of decreasing $Deg_v(q)$. In fact it can be proven that after this procedure we always have $Deg_v(q) < Deg_v(p)$.

It is interesting to note that the $R2$ algorithm reduces to the well known algorithm of Euclid to compute the GCD of $p$ and $q$, in the case these polynomials are univariate.

As in our case we only have two starting polynomials $F_1$ and $F_2$, it is more efficient to consider $R2$ than $NF$. Moreover when using $R2$ we do not have to compute S-polynomials for two polynomials whose $Deg_v$ are the same ($R2$ eliminates these cases, see example above). Let us now consider the computation of a S-polynomial, i.e $p := SP(F_i, F_j)$, with $Deg_v(F_i) = Deg_v(F_j) + 1$ in this case we obtain $Deg_v(p) \leq Deg_v(F_i)$. If we compute $R2(F_i, p)$ and then $R2(F_j, p)$ we will have $Deg_v(p) < Deg_v(F_j) < Deg_v(F_i)$. Now, if we put $F_k = p$ then we can do the same as above with $F_j$ and $F_k$, to obtain a polynomial $F_l$. As $Deg_v$ decrease each time, it will necesserally become zero, in this last case we get a univariate polynomial in $u$. By using this ideas we can formulate the following simpler version of the GB0 algorithm:

$R2(F_1, F_2)$;
$i := 1$;

while $F_{i+1} \neq 0$ do

    $F_{i+2} := SP(F_i, F_{i+1})$;
    $R2(F_i, F_{i+2})$;
    $R2(F_{i+1}, F_{i+2})$;
    $i := i + 1$;

end while

At the end of this algorithm the Groebner basis is thus given by $G = \{F_1, \ldots, F_N\}$, with $N \leq m$. If $F_N$ is equal to 1 then there is no intersection and we stop, else $F_N$ is a univariate polynomial in $u$, which must be solved.

## 4.3 Solving the univariate polynomial in u

Several methods exist to solve polynomial equations. Here again we will consider an algebraic approach which gives us all the roots without having to worry about initial guesses (as for the Newton iteration).

First we have to compute a sequence of disjoint intervals each containing exactly one root of the polynomial equation. If none of these exist then the equation has no real roots and we stop. This is a particulary useful criterium when the surface is defined on a limited parameter space, for example if the parameter space is $(u,v) \in [0,1] \times [0,1]$ then we consider only those intervals within $[0,1]$. A large number of algorithms exists to compute this sequence of intervals [4]. Currently Sturm's method has been implemented for its simplicity. Another interesting method is the modified Uspensky algorithm by Collins [4], which has been used successfully by Hanrahan to ray-trace implicit algebraic surfaces [5].

Once we have computed the sequence of intervals we have to search for the root each one contains. Here again we have a large choice of methods, from the simple bisection method to the Laguerre iteration used by Kajiya [6]. As we want to keep things simple for the moment, the bisection method is used. At the end of this step we thus get a set $\{u_1, \ldots, u_s\}$ of roots.

### 4.4 Calculating the actual intersection

At this stage we must not forget that we have a Groebner basis $\{F_1, \ldots, F_N\}$, in particular $F_{N-1}$ has the smallest non zero $Deg_v$. Now if we substitute $u$ in $F_{N-1}$ with an $u_i$ then we get a univariate polynomial in $v$ of degree $Deg_v(F_{N-1})$. In most cases this degree will be one, so that we can calculate the corresponding $v_i$ directly.

For each intersection point $(u_i, v_i)$ we calculate the corresponding $t_i$ with equation 5 (if $V^x \neq 0$). Then we select the smallest positive $t_i$. If there is none then the ray misses the surface, otherwise the $t$ value is the nearest intersection point of the ray with the surface.

## 5 Results

The algorithm described in the previous section has been implemented in C on a SUN 4 work station. We chose Bezier patches as examples of polynomial patches so as to have a standard reference. Figures 1, 2 and 3 are such patches ray traced with our algorithm.

As we work with floating point numbers, we have to be very careful about numerical instabilities, especially when computing the Groebner basis, which involves many substractions. One simple heuristic to avoid these instabilities is to consider small values as zero. In the current implementation, numbers were considered as zero if smaller than $10^9$ in absolute value.

As stated in the introduction one major drawback of the algebraic approach of Kajiya was the high degree of the univariate polynomial in $u$, which cannot be solved without numerical instabilities. Therefore it is interesting to provide for each image the degree of the resulting polynomial in $u$, and to add the size



of the Groebner basis and of the polynomial in $v$. All these values were the same for each ray in the image. The size of each image is $840 \times 840$.

| *Figure* | *Size of Groebner basis* | *Degree of polynomial in u* | *Degree of polynomial in v* |
|---|---|---|---|
| 1 | 3 | 2 | 1 |
| 2 | 2 | 4 | 1 |
| 3 | 2 | 6 | 1 |

Kajiya proved in [6], using Bezout's theorem, that in the case of a polynomial patch with $n = m = 3$, the degree of the resulting polynomial in $u$ is at most 18. This upper bound is clearly not attained in our examples; this is unfortunately not due to our algorithm but rather to the type of surfaces used. When applying our algorithm to more general patches, we had some nice views of numerical instabilities ! This was due mainly to our implementation of the Sturm algorithm, which became instable for polynomials of degree higher than 8.

## 6 Conclusion

We have presented in this paper a new method to ray trace polynomial patches using an algebraic approach. The results, however, prove that the method is no great improvement over existing ones.

It seems that parametric surfaces, which are widely used in wire frame environments, are very difficult to ray trace with a direct numerical method, as opposed to implicit surfaces. The most popular methods existing modify the parametric representation: Snyder and Barr in [8] first tesselate the surface before ray tracing it, Sederberg and Anderson in [7] use implicitization for a special case: the Steiner patch.

The great number of papers published on the problem of ray tracing parametric surfaces is an indication that the problem is a difficult one. Future work in this area should probably concentrate on new representations of surfaces more suitable for ray tracing.

## References


[1] B. Buchberger, *Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems*, Aequationes Math. 4 (1970), pages 374-383.

[2] B. Buchberger, *A Criterion for Detecting Unnecessary Reductions in the Construction of Groebner Bases*, EUROSAM 1979, Lecture Notes in Computer Science, Vol. 72, Berlin-Heidelberg-New York: Springer 1979, pages 3-21.

[3] B. Buchberger, *Groebner Bases: An Algorithmic Method in Polynomial Ideal Theory*, Multidimensional Systems Theory, D. Reidel Publishing Company, 1985, pages 184-232.

[4] G. Collins, and R. Loos, *Real Zeros of Polynomials*, Computing, Suppl. 4, Computer Algebra : Symbolic and Algebraic Computation, Springer-Verlag 1982, pages 83-94.

[5] P. Hanrahan, *Ray Tracing Algebraic Surfaces*, Computer Graphics (Proc. SIGGRAPH 83), Vol.17, No. 3, July 1983, pages 137-144.

[6] J. Kajiya, *Ray Tracing Parametric Patches*, Computer Graphics (Proc. SIGGRAPH 82), Vol. 16, No. 3, July 1982, pages 245-254.

[7] T.W. Sederberg, and D.C. Anderson, *Ray Tracing of Steiner Patches*, Computer Graphics (Proc. SIGGRAPH 84), Vol. 18, No. 3, July 1984, pages 159-164.

[8] J. Snyder, and A. Barr, *Ray Tracing Complex Models Containing Surface Tesselations*, Computer Graphics (Proc. SIGGRAPH 87), Vol. 21, No. 4, July 1987, pages 119-128.

[9] M. Sweeney, and R. Bartels, *Ray Tracing Free-Form B-Spline Surfaces*, IEEE Computer Graphics and Applications 6(2), February 1986, pages 41-49.

[10] D. Toth, *On Ray Tracing Parametric Surfaces*, Computer Graphics (Proc. SIGGRAPH 85), Vol. 19, No. 3, July 1985, pages 171-179.

[11] T. Whitted, *An Improved Model For Shaded Display*, Comm. ACM, Vol. 23, No. 6, June 1980, pages 343-349.



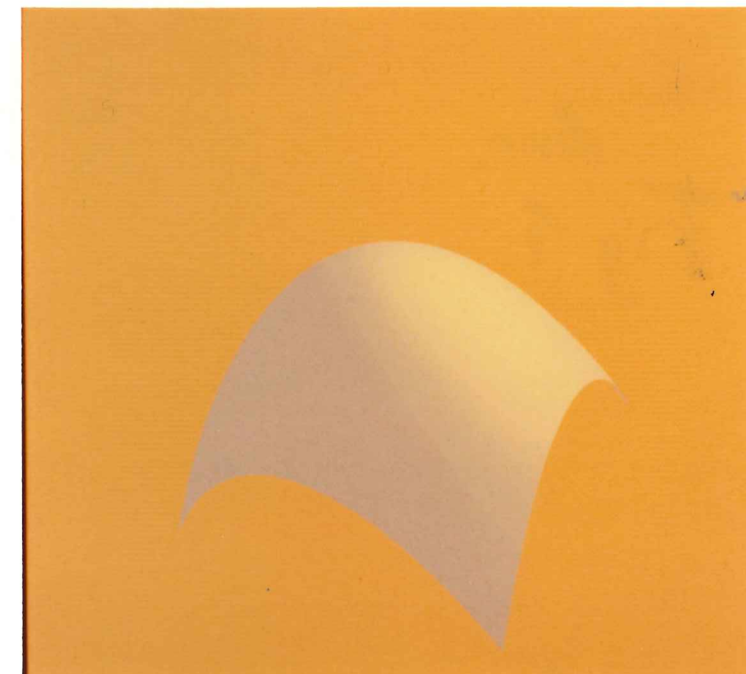

Figure 1: bezier patch number 1

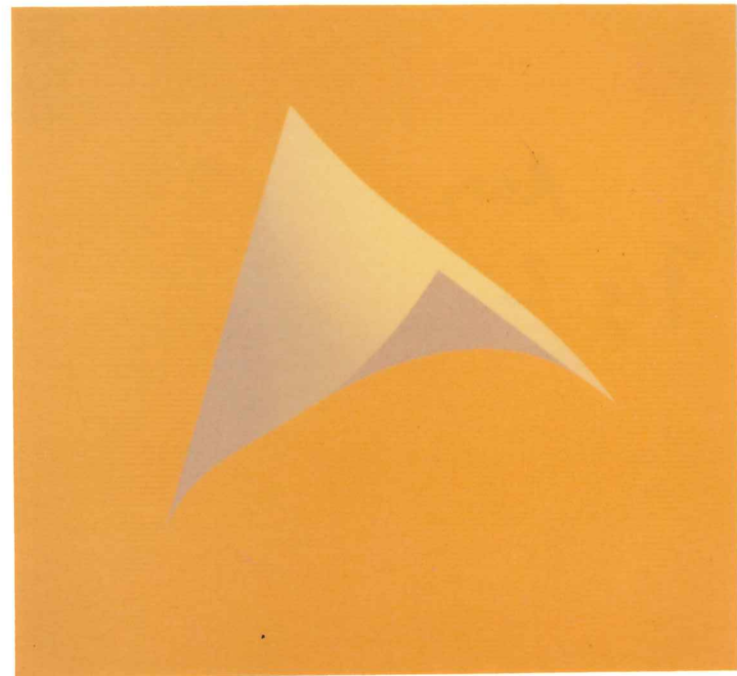

Figure 2: bezier patch number 2



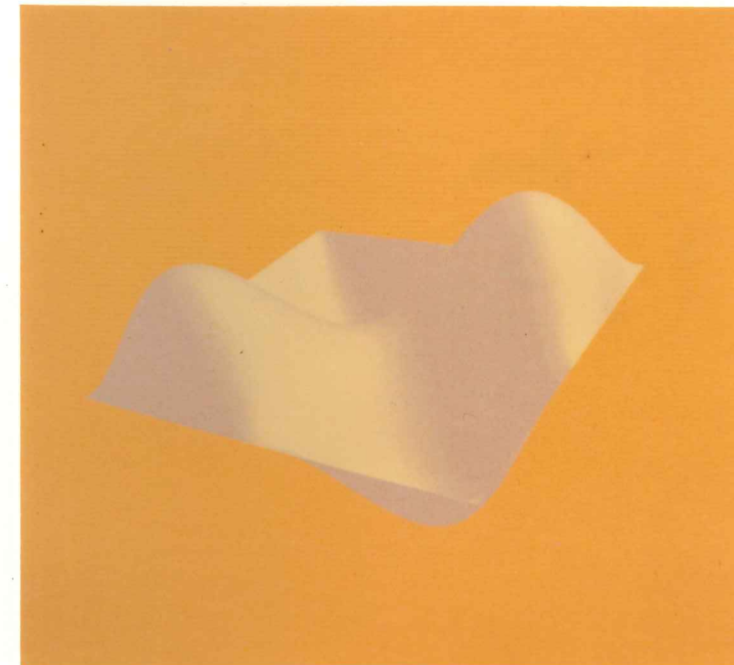

Figure 3: bezier patch number 3

# Annexe B

# Code source

36



```
/*
  ==========================================================================
   RAY TRACER --- ray.h
  --------------------------------------------------------------------------
   Content       : global constants, types and macros for the ray tracer
   Author        : Stam Jos
   Creation Date : 06/02/88
   Last Modified : 17/08/88
  ==========================================================================
*/


/*************************** CONSTANTS DEFINITION ***************************/

#define BIG   1.0e10
#define SMALL 1.0e-3
#define PRECISION 1.0e-9

#define NBPOL   20
#define NBMON   200
#define MAX_DEG 40

#define DEG_TO_RAD 0.01745329252

#ifndef NULL
#define NULL 0
#endif

#define FALSE 0
#define TRUE  !FALSE

#define MAX_NB_OBJECTS 20
#define MAX_NB_LIGHTS  10
#define MAX_NB_RECUR   5
#define MAX_NB_CHAR    80

#define BLACKBODY   0
#define AMBIANT     10
#define DIFFUSE     20
#define TEXTURE     25
#define SPECULAR    30
#define MIRROR      40
#define TRANSPARENT 50

/***************************** TYPES DEFINITION *****************************/

typedef int BOOLEAN;

typedef struct { double X, Y, Z; } POINT;
typedef struct { double X, Y, Z; } VECTOR;

typedef struct { double R, G, B; } COLOR;

typedef struct { COLOR Cs, Cr;
                 double Ka, Kd, Ks, Kt;
                 double n, Kn; } ILLUM;

typedef struct { POINT  Org;
                 VECTOR Dir;
                 short  RecLevel; } RAY;

typedef struct { short  ObjNum;
                 double Dist;
                 POINT  Inters;
```

```
                      VECTOR Normal; } PATCH;

typedef struct { POINT C; double R2;} SPHERE;

typedef struct { POINT Org; VECTOR Normal; double d; } PLANE;

typedef struct { double Xmin, Xmax, Ymin, Ymax, Zmin, Zmax; } BOX;

typedef struct { POINT  a, d;
                 VECTOR ab,  ac,  db,  dc;
                 PLANE  Pl; } POLYGON;

typedef double SPLINE[4][4][3];

typedef struct { POINT Source;
                 COLOR Col; } LIGHT;

typedef struct { short IllumId;
                 void  *Param;
                 void  *Bound;
                 ILLUM Ill;
                 BOOLEAN (*BoundRoutine)();
                 BOOLEAN (*IntersRoutine)();
                 void (*NormalRoutine)(); } OBJECT;

typedef struct { double Focal, Dist, VField,
                        Balance, LeftRight, TopBottom;
                 POINT Center; } CAMERA;

typedef struct { COLOR BackGnd;
                 short NbObjects, NbLights;
                 BOOLEAN Shadows;
                 OBJECT Objects[ MAX_NB_OBJECTS ];
                 LIGHT  Lights[ MAX_NB_LIGHTS ]; } WORLD;

typedef struct { int SizeX, SizeY, OrgX, OrgY, ResX, ResY; } VIEWPORT;

typedef struct { double a[MAX_DEG]; int deg; } STURM_SERIE;

typedef struct { double a, b; } INTERVAL;


/*************************** MACROS DEFINITION ***************************/

#define DOT(u,v)            ( (u).X*(v).X + (u).Y*(v).Y + (u).Z*(v).Z )
#define NORM2(v)            DOT(v,v)
#define NORM(v)             sqrt(DOT(v,v))
#define NORMALIZE(v)        { double nnn=NORM(v); \
                              (v).X /= nnn; (v).Y /= nnn; (v).Z /= nnn; }

#define MOVE(u,v)           { (u).X=(v).X; (u).Y=(v).Y; (u).Z=(v).Z; }
#define ADD(u,v,w)          { (u).X = (v).X + (w).X; \
                              (u).Y = (v).Y + (w).Y; \
                              (u).Z = (v).Z + (w).Z; }

#define SUB(u,v,w)          { (u).X = (v).X - (w).X; \
                              (u).Y = (v).Y - (w).Y; \
                              (u).Z = (v).Z - (w).Z; }

#define PROD(u,v,w)         { (u).X = (v).Y*(w).Z - (v).Z*(w).Y; \
                              (u).Y = (v).Z*(w).X - (v).X*(w).Z; \
                              (u).Z = (v).X*(w).Y - (v).Y*(w).X; }

#define GETPOINT(p,o,t,d)   { (p).X = (o).X + (t)*(d).X; \
                              (p).Y = (o).Y + (t)*(d).Y; \
                              (p).Z = (o).Z + (t)*(d).Z; }

#define MAX(a,b)            ( (a) > (b) ? (a) : (b) )
#define MIN(a,b)            ( (a) < (b) ? (a) : (b) )
```



```
#define ABS(x)              ( (x) > 0.0 ? (x) :-(x) )
#define ZERO(a)             ( ((a)<PRECISION) && ((a)>-PRECISION) )
```

```
/*
  ============================================================================
   RAY TRACER --- ray0.c
  ----------------------------------------------------------------------------
   Content: main module and basic ray tracing logic
   Author : Stam Jos
   Creation Date : 06/02/88
   Last Modified : 17/08/88
  ============================================================================
*/

#include <stdio.h>
#include "ray.h"


/*************************** EXTERNAL REFERENCES ***************************/

extern double  sqrt(), sin(), cos(), tan(), pow();
extern BOOLEAN InitScreen(), InitScene();
extern void    Plot(), CleanUp();

extern int SizeTexture, SizeTextureX, SizeTextureY;
extern double UnitTexture;
extern unsigned char *TextureR, *TextureG, *TextureB;

extern int DegPolU[MAX_DEG], DegPolV[MAX_DEG], GroebnerSize[MAX_DEG], NbCrit;


/************************** MODULES DEFINITION ****************************/

/*
  ============================================================================
   ProjectPlane --- compute projection plane
  ============================================================================
*/

void ProjectPlane( c, vp, oc, o, ir, ic)
CAMERA   *c;
VIEWPORT *vp;
POINT    *oc, *o;
VECTOR   *ir, *ic;
{
   double bc, c1, c2, c3, s1, s2, s3,
          a11, a12, a13, a21, a22, a23, a31, a32, a33;

   bc = c->Focal * tan( c->VField / 2.0 * DEG_TO_RAD ) / sqrt(2.0);

   c1 =cos( c->Balance   * DEG_TO_RAD ); s1 =sin( c->Balance   * DEG_TO_RAD );
   c2 =cos(-c->TopBottom * DEG_TO_RAD ); s2 =sin(-c->TopBottom * DEG_TO_RAD );
   c3 =cos(-c->LeftRight * DEG_TO_RAD ); s3 =sin(-c->LeftRight * DEG_TO_RAD );

   a11 = c1*c3 + s1*s2*s3  ; a12 = -c3*s1 + c1*s2*s3 ; a13 = c2*s3 ;
   a21 = c2*s1             ; a22 = c1*c2             ; a23 = -s2   ;
   a31 = -c1*s3 + c3*s1*s2 ; a32 = s1*s3 + c1*c3*s2  ; a33 = c2*c3 ;

   oc->X = a13*(c->Dist+c->Focal) + c->Center.X;
   oc->Y = a23*(c->Dist+c->Focal) + c->Center.Y;
   oc->Z = a33*(c->Dist+c->Focal) + c->Center.Z;

   o->X = -a11*bc - a12*bc + a13*c->Dist;
   o->Y = -a21*bc - a22*bc + a23*c->Dist;
   o->Z = -a31*bc - a32*bc + a33*c->Dist;

   ir->X = vp->ResX * ( ( a11*bc - a12*bc + a13*c->Dist) - o->X ) / vp->SizeX;
```

```
   ir->Y = vp->ResX * ( ( a21*bc - a22*bc + a23*c->Dist) - o->Y ) / vp->SizeX;
   ir->Z = vp->ResX * ( ( a31*bc - a32*bc + a33*c->Dist) - o->Z ) / vp->SizeX;

   ic->X = ( (-a11*bc + a12*bc + a13*c->Dist) - o->X ) / vp->SizeY;
   ic->Y = ( (-a21*bc + a22*bc + a23*c->Dist) - o->Y ) / vp->SizeY;
   ic->Z = ( (-a31*bc + a32*bc + a33*c->Dist) - o->Z ) / vp->SizeY;

   o->X += c->Center.X;
   o->Y += c->Center.Y;
   o->Z += c->Center.Z;

   return;
}


/*
  ==========================================================================
   IntSphere --- calculate intersection of ray with sphere (if any)
  ==========================================================================
*/

BOOLEAN IntSphere ( ray, sphere, patch, objnum, shadow )
RAY    *ray;
SPHERE *sphere;
PATCH  *patch;
short   objnum;
BOOLEAN shadow;
{
   VECTOR CO;
   double b, c, d, t;

   SUB( CO, ray->Org, sphere->C );
   b = DOT( ray->Dir, CO );
   c = NORM2(CO) - sphere->R2;
   if ( (d = b*b -c) < SMALL ) return FALSE;
   d = sqrt(d);
   if ( (t = -(b+d)) > SMALL )
   {
      if ( t >= patch->Dist ) return TRUE;
      patch->Dist = t;
      if ( shadow ) return TRUE;
      patch->ObjNum = objnum;
      GETPOINT( patch->Inters, ray->Org, t, ray->Dir );
      return TRUE;
   }
   if ( (t = -b+d) > SMALL )
   {
      if ( t >= patch->Dist ) return TRUE;
      patch->Dist = t;
      if ( shadow ) return TRUE;
      patch->ObjNum = objnum;
      GETPOINT( patch->Inters, ray->Org, t, ray->Dir );
      return TRUE;
   }
   return FALSE;
}


/*
  ==========================================================================
   IntPlane --- calculate intersection of ray with plane (if any)
  ==========================================================================
*/
```



```
BOOLEAN IntPlane( ray, plane, patch, objnum, shadow )
RAY    *ray;
PLANE  *plane;
PATCH  *patch;
short   objnum;
BOOLEAN shadow;
{
   double dp, t;

   dp = DOT( ray->Dir, plane->Normal);
   if ( dp < SMALL && dp > -SMALL ) return FALSE;
   t = - (plane->d + DOT(ray->Org,plane->Normal)) / dp;
   if ( t < SMALL || t >= patch->Dist) return FALSE;
   patch->Dist = t;
   if ( shadow ) return TRUE;
   patch->ObjNum = objnum;
   GETPOINT( patch->Inters, ray->Org, t, ray->Dir );
   return TRUE;
}


/*
  ==========================================================================
   InQuartPlane --- determine if given vector is in the given 1/4 plane
  ==========================================================================
*/

BOOLEAN InQuartPlane( x, u, v )
VECTOR *x, *u, *v;
{
   double g;

   if ( (g=u->Y*v->X-u->X*v->Y) != 0.0 )
      return g*(u->Y*x->X-u->X*x->Y) >= 0.0 && g*(v->X*x->Y-v->Y*x->X) >= 0.0;
   if ( (g=u->Z*v->X-u->X*v->Z) != 0.0 )
      return g*(u->Z*x->X-u->X*x->Z) >= 0.0 && g*(v->X*x->Z-v->Z*x->X) >= 0.0;
   if ( (g=u->Z*v->Y-u->Y*v->Z) != 0.0 )
      return g*(u->Z*x->Y-u->Y*x->Z) >= 0.0 && g*(v->Y*x->Z-v->Z*x->Y) >= 0.0;
   return FALSE;
}


/*
  ==========================================================================
   IntPolygon --- calculate intersection of ray with polygon (if any)
  ==========================================================================
*/

BOOLEAN IntPolygon( ray, pol, patch, objnum, shadow )
RAY     *ray;
POLYGON *pol;
PATCH   *patch;
short    objnum;
BOOLEAN  shadow;
{
   VECTOR ax, dx;
   PATCH  patch0;

   patch0.Dist = patch->Dist;
   if ( ! IntPlane( ray, &(pol->Pl), &patch0, objnum, FALSE ) ) return FALSE;

   SUB( ax, patch0.Inters, pol->a );
   if ( ! InQuartPlane( &(ax), &(pol->ab), &(pol->ac) ) ) return FALSE;
   SUB( dx, patch0.Inters, pol->d );
```

```
    if ( ! InQuartPlane( &(dx), &(pol->db), &(pol->dc) ) ) return FALSE;

    patch->Dist = patch0.Dist;
    if ( shadow ) return TRUE;
    patch->ObjNum = patch0.ObjNum;
    MOVE( patch->Inters,  patch0.Inters );
    return TRUE;
}


/*
  ==============================================================================
   NormSphere --- calculate normal vector for a given point on a sphere
  ==============================================================================
*/

void NormSphere ( sphere, patch )
SPHERE *sphere;
PATCH  *patch;
{
    double r = sqrt(sphere->R2);

    patch->Normal.X = ( patch->Inters.X - sphere->C.X) / r;
    patch->Normal.Y = ( patch->Inters.Y - sphere->C.Y) / r;
    patch->Normal.Z = ( patch->Inters.Z - sphere->C.Z) / r;

    return;
}


/*
  ==============================================================================
   NormPlane --- calculate normal vector for given point on plane
  ==============================================================================
*/

void NormPlane ( plane, patch )
PLANE *plane;
PATCH *patch;
{
    MOVE( patch->Normal, plane->Normal );
    return;
}


/*
  ==============================================================================
   NormPolygon --- calculate normal vector for a given point on a polygon
  ==============================================================================
*/

void NormPolygon ( pol, patch )
POLYGON *pol;
PATCH   *patch;
{
    MOVE( patch->Normal, pol->Pl.Normal );
    return;
}


/*
  ==============================================================================
   IntBoundBox --- get intersection of ray with bounding box (if any)
  ==============================================================================
```



```
*/

BOOLEAN IntBoundBox( ray, box, t1, t2 )
RAY    *ray;
BOX    *box;
double *t1, *t2;
{
   double xt, tx, yt, ty, zt, tz, o, d;

   o = ray->Org.X;
   if ( (d = ray->Dir.X) != 0 )
   {
      xt = (box->Xmin-o) / d;
      tx = (box->Xmax-o) / d;
      if ( xt > tx ) { d=xt; xt=tx; tx=d; }
   }
   else
   {
      if ( o < box->Xmin || o > box->Xmax ) return FALSE;
      xt = SMALL; tx = BIG;
   }

   o = ray->Org.Y;
   if ( (d = ray->Dir.Y) != 0 )
   {
      yt = (box->Ymin-o) / d;
      ty = (box->Ymax-o) / d;
      if ( yt > ty ) { d=yt; yt=ty; ty=d; }
   }
   else
   {
      if ( o < box->Ymin || o > box->Ymax ) return FALSE;
      yt = SMALL; ty = BIG;
   }

   o = ray->Org.Z;
   if ( (d = ray->Dir.Z) != 0 )
   {
      zt = (box->Zmin-o) / d;
      tz = (box->Zmax-o) / d;
      if ( zt > tz ) { d=zt; zt=tz; tz=d; }
   }
   else
   {
      if ( o < box->Zmin || o > box->Zmax ) return FALSE;
      zt = SMALL; tz = BIG;
   }

   *t1 = MAX( MAX(xt,yt), zt );
   *t2 = MIN( MIN(tx,ty), tz );
   return *t1 <= *t2 && *t2 >= SMALL;
}


/*
  ==============================================================================
   GetIntersection --- get nearest intersection of given ray
  ==============================================================================
*/

BOOLEAN GetIntersection ( ray, world, patch, shadow )
RAY    *ray;
WORLD  *world;
PATCH  *patch;
```

```
BOOLEAN shadow;
{
   OBJECT *o;
   short i;
   double t1, t2;

   patch->Dist = BIG;
   for ( i = 0 ; i < world->NbObjects ; i++ )
   {
      o = &world->Objects[i];
      t1 = SMALL;
      t2 = BIG;
      if ( o->Bound != NULL )
      {
         if (  ! (*(o->BoundRoutine))( ray, o->Bound, &t1, &t2 ) ) continue;
         t1 = MAX( SMALL, t1 );
      }
      (*(o->IntersRoutine))( ray, o->Param, patch, i, shadow, t1, t2 );
   }
   if ( patch->Dist == BIG ) return FALSE;
   if ( shadow ) return TRUE;
   o = &(world->Objects[ patch->ObjNum ]);
   (*(o->NormalRoutine))( o->Param, patch );
   return TRUE;
}


/*
  ==============================================================================
   IllumModel --- calculates illumination value at given intersection
  ==============================================================================
*/

void IllumModel ( ray, patch, l, o, world, color )
RAY    *ray;
PATCH  *patch;
LIGHT  *l;
OBJECT *o;
WORLD  *world;
COLOR  *color;
{
   VECTOR  L, R;
   RAY     ray0;
   PATCH   patch0;
   ILLUM  *ill = &(o->Ill);
   double  cos_alpha, cos_theta, less, distL, distR, i;

   SUB( L, l->Source, patch->Inters );
   distL = NORM( L );
   NORMALIZE( L );
   if ( (cos_theta = DOT(patch->Normal,L)) <= 0.0 ) return;
   if ( ray->RecLevel > 1 ) distR = 0.0 /*patch->Dist*/;
   else                     distR = 0.0;
   less = 5.0 / (5.0+distL+distR);
   MOVE( ray0.Org, patch->Inters );
   MOVE( ray0.Dir, L );
   if ( world->Shadows )
   {
      if ( GetIntersection( &ray0, world, &patch0, TRUE ) )
         if ( patch0.Dist < distL ) return;
   }
   i = cos_theta*ill->Kd*less;
   color->R += i*ill->Cs.R*l->Col.R;
   color->G += i*ill->Cs.G*l->Col.G;
```



```
   color->B += i*ill->Cs.B*l->Col.B;
   if ( o->IllumId < SPECULAR ) return;
   GETPOINT( R, L, -2.0*cos_theta, patch->Normal );
   if ( (cos_alpha = DOT(R,ray->Dir)) <= 0.0 ) return;
   i = pow( cos_alpha, ill->n ) * ill->Ks * less;
   color->R += i*ill->Cr.R*l->Col.R;
   color->G += i*ill->Cr.G*l->Col.G;
   color->B += i*ill->Cr.B*l->Col.B;
   return;
}


/*
  ==============================================================================
   ColTexture --- change color of surface of plane according to texture
  ==============================================================================
*/

void ColTexture ( ill, p, patch )
ILLUM  *ill;
PLANE  *p;
PATCH  *patch;
{
   VECTOR v;
   int i;

   SUB(v,patch->Inters,p->Org);
   i  = (((int)(ABS(v.Y)/UnitTexture*SizeTextureY))%SizeTextureY)*SizeTextureX;
   i +=  ((int)(ABS(v.X)/UnitTexture*SizeTextureX))%SizeTextureX;
   ill->Cs.R = *(TextureR+i)/255.0;
   ill->Cs.G = *(TextureG+i)/255.0;
   ill->Cs.B = *(TextureB+i)/255.0;
   return;
}


/*
  ==============================================================================
   Intensity --- calculates intensity of point at intersection
  ==============================================================================
*/

void Intensity ( ray, patch, world, color )
RAY   *ray;
PATCH *patch;
WORLD *world;
COLOR *color;
{
   OBJECT *o = &(world->Objects[patch->ObjNum]);
   VECTOR  L;
   RAY     ray0;
   COLOR   color0;
   double  cos_alpha = DOT(patch->Normal,ray->Dir), kn, p;
   short   i;
   BOOLEAN InsideObject;
   void    TraceRay();

   if ( o->IllumId == TEXTURE ) ColTexture( &(o->Ill), o->Param, patch );
   InsideObject = FALSE;
   if ( cos_alpha > 0 )
   {
      cos_alpha = -cos_alpha;
      patch->Normal.X = -patch->Normal.X;
      patch->Normal.Y = -patch->Normal.Y;
```

```
         patch->Normal.Z = -patch->Normal.Z;
         InsideObject = TRUE;
      }
      if ( o->IllumId >= AMBIANT )
      {
         color->R += o->Ill.Cs.R * o->Ill.Ka;
         color->G += o->Ill.Cs.G * o->Ill.Ka;
         color->B += o->Ill.Cs.B * o->Ill.Ka;
      }
      if ( o->IllumId >= DIFFUSE )
      {
         for ( i = 0 ; i < world->NbLights ; i++ )
         {
            IllumModel(ray,patch,&(world->Lights[i]),o,world,color);
         }
      }
      if ( o->IllumId >= MIRROR )
      {
         MOVE( ray0.Org, patch->Inters );
         p = -2.0*cos_alpha;
         GETPOINT( ray0.Dir, ray->Dir, p, patch->Normal );
         ray0.RecLevel = ray->RecLevel;
         color0.R = color0.G = color0.B = 0.0;
         TraceRay( &ray0, world, &color0 );
         color->R += o->Ill.Ks*color0.R;
         color->G += o->Ill.Ks*color0.G;
         color->B += o->Ill.Ks*color0.B;
      }
      if ( o->IllumId >= TRANSPARENT )
      {
         MOVE( ray0.Org, patch->Inters );
         if ( InsideObject ) kn = 1.0 / o->Ill.Kn;
         else                kn = o->Ill.Kn;
         p = 1.0 - (1.0-cos_alpha*cos_alpha) / (kn*kn);
         if ( p >= 0 )
         {
            p = -cos_alpha/kn - sqrt(p);
            ray0.Dir.X = ray->Dir.X/kn + p*patch->Normal.X;
            ray0.Dir.Y = ray->Dir.Y/kn + p*patch->Normal.Y;
            ray0.Dir.Z = ray->Dir.Z/kn + p*patch->Normal.Z;
            ray0.RecLevel = ray->RecLevel;
            color0.R = color0.G = color0.B = 0.0;
            TraceRay( &ray0, world, &color0 );
            color->R += o->Ill.Kt*color0.R;
            color->G += o->Ill.Kt*color0.G;
            color->B += o->Ill.Kt*color0.B;
         }
      }
      color->R = color->R > 1.0 ? 1.0 : color->R;
      color->G = color->G > 1.0 ? 1.0 : color->G;
      color->B = color->B > 1.0 ? 1.0 : color->B;
      return;
}


/*
  ============================================================================
    BackGnd --- returns background intensity for given ray
  ============================================================================
*/

void BackGnd ( ray, world, color )
RAY   *ray;
WORLD *world;
```



```
COLOR *color;
{
   color->R += world->BackGnd.R;
   color->G += world->BackGnd.G;
   color->B += world->BackGnd.B;
   return;
}


/*
  ============================================================================
   TraceRay --- return value of intensity for given ray
  ============================================================================
*/

void TraceRay ( ray, world, color )
RAY   *ray;
WORLD *world;
COLOR *color;
{
   PATCH patch;
   OBJECT *o;

   if ( ++ray->RecLevel >= MAX_NB_RECUR ) return;
   if ( GetIntersection( ray, world, &patch, FALSE ) )
      Intensity( ray, &patch, world, color );
   else
      BackGnd( ray, world, color );
   return;
}


/*
  ============================================================================
   TraceScene --- do the ray tracing for the given scene
  ============================================================================
*/

void TraceScene ( world, camera, viewport )
WORLD    *world;
CAMERA   *camera;
VIEWPORT *viewport;
{
   POINT  OptCenter, Origin;
   VECTOR IncRow, IncCol;
   COLOR  color;
   RAY    ray;
   int    x, y;

   if ( ! world->NbObjects) return;
   ray.RecLevel = 0;
   ProjectPlane( camera, viewport, &OptCenter, &Origin, &IncRow, &IncCol );
   Origin.X += ( IncRow.X + IncCol.X) / 2.0;
   Origin.Y += ( IncRow.Y + IncCol.Y) / 2.0;
   Origin.Z += ( IncRow.Z + IncCol.Z) / 2.0;

   for ( y = 0 ; y < viewport->SizeY ; y += viewport->ResY )
   {
      int y0 = viewport->SizeY - y - 1;

      GETPOINT( ray.Org, Origin, y, IncCol );
      for ( x = 0 ; x < viewport->SizeX ; x += viewport->ResX )
      {
         SUB( ray.Dir, ray.Org, OptCenter );
```

```
         NORMALIZE( ray.Dir );
         ray.RecLevel = 0;
         color.R = color.G = color.B = 0.0;
         TraceRay( &ray, world, &color );
         ADD( ray.Org, ray.Org, IncRow );
         Plot( viewport, x, y0, &color);
      }
   }
   return;
}


/*
  ============================================================================
   main --- main module of the program
  ============================================================================
*/

main( argc, argv )
int     argc;
char    *argv[];
{
   WORLD    world;
   CAMERA   camera;
   VIEWPORT viewport;
   int      i;

   if ( argc != 2 )
   {
      puts( "bad arguments\n" );
      exit( 0 );
   }

   NbCrit = 0;
   for ( i=0 ; i<MAX_DEG ; i++ )
   {
      DegPolU[i] = DegPolV[i] = GroebnerSize[i] = 0;
   }
   if ( ! InitScene( *++argv, &world, &camera, &viewport ) )
   {
      puts( "cannot initialise scene with given data\n");
      exit( 0 );
   }
   if ( ! InitScreen( &viewport ) )
   {
      puts( "cannot initialise screen\n");
      exit(0);
   }

   TraceScene( &world, &camera, &viewport );

   printf("\nSTATISTICS:\n");
   printf("\ncriterium C1 applied %d times\n", NbCrit );
   printf("\nGroebner basis size:\n");
   for ( i=0 ; i<MAX_DEG ; i++ )
   {
      if ( GroebnerSize[i] != 0 )
         printf("N=%d : %d\n", i, GroebnerSize[i] );
   }
   printf("\nDegrees of polynomials in u:\n");
   for ( i=0 ; i<MAX_DEG ; i++ )
   {
      if ( DegPolU[i] != 0 )
      printf("d=%d : %d\n", i, DegPolU[i] );
```



```
   }
   printf("\nDegrees of polynomials in v:\n");
   for ( i=0 ; i<MAX_DEG ; i++ )
   {
      if ( DegPolV[i] != 0 )
      printf("d=%d : %d\n", i, DegPolV[i] );
   }
#ifdef SUN_DISPLAY
   getchar();
#endif
   CleanUp( &viewport );
}
```

```
/*
  ==============================================================================
   RAY TRACER --- ray1.c
  ------------------------------------------------------------------------------
   Content       : Display and I/O modules for ray tracer
   Author        : Stam Jos
   Creation Date : 06/02/88
   Last Modified : 29/08/88
  ==============================================================================
*/

#include <stdio.h>

#ifdef SUN_DISPLAY
#include <suntool/sunview.h>
#include <suntool/panel.h>
#include <suntool/canvas.h>
#endif

#include "ray.h"


/*************************** GLOBAL VARIABLES DEFINITION **********************/

#ifdef SUN_DISPLAY
static unsigned char ColorMap[3][256];
static Frame   SUN_Frame;
static Canvas  SUN_Canvas;
static Pixwin *SUN_PixWin;
#endif

static char SaveFile[ MAX_NB_CHAR ];
static BOOLEAN SaveFlag;
static char *SaveR, *SaveG, *SaveB;
static char DumpFile[ MAX_NB_CHAR ];
static BOOLEAN DumpFlag;

int SizeTexture, SizeTextureX, SizeTextureY;
double UnitTexture;
unsigned char *TextureR, *TextureG, *TextureB;


/****************************** MODULES DEFINITION **************************/

extern BOOLEAN IntSphere(), IntPlane(), IntPolygon();
extern BOOLEAN IntAlgSurf(), IntSpline();
extern void NormSphere(), NormPlane(), NormPolygon();
extern void NormAlgSurf(), NormSpline();
extern BOOLEAN IntBoundBox();
extern void *malloc();
extern double sqrt();


/*
  ==============================================================================
   InitScreen --- initialises a screen for display (SUN specific)
  ==============================================================================
*/

BOOLEAN InitScreen ( vp )
VIEWPORT *vp;
{
   int i;
```

```
#ifdef SUN_DISPLAY
   SUN_Frame = window_create( NULL, FRAME,
                              FRAME_LABEL, "Ray Tracer",
                              0 );
   SUN_Canvas = window_create( SUN_Frame, CANVAS,
                               CANVAS_RETAINED, TRUE,
                               WIN_WIDTH,  vp->SizeX,
                               WIN_HEIGHT, vp->SizeY,
                               0 );
   SUN_PixWin = canvas_pixwin( SUN_Canvas );
   pw_setcmsname( SUN_PixWin, "pxwn" );
   for ( i = 0 ; i < 256 ; i++ )
   {
      ColorMap[0][i] = i;
      ColorMap[1][i] = i;
      ColorMap[2][i] = i;
   }
   pw_putcolormap( SUN_PixWin, 0, 256, ColorMap[0], ColorMap[1], ColorMap[2] );
   window_fit( SUN_Frame );
   window_set( SUN_Frame, WIN_SHOW, TRUE, 0 );
   (void)notify_dispatch();
#endif

   return TRUE;
}


/*
  ==============================================================================
   Plot --- display a given pixel with a given color (SUN specific)
  ==============================================================================
*/

void Plot ( vp, x, y, color )
VIEWPORT *vp;
int       x, y;
COLOR    *color;
{
   unsigned int col = (unsigned int)
                      ((0.3*color->R+0.59*color->G+0.11*color->B)*255);

   if ( SaveFlag )
   {
      *(SaveR+y*vp->SizeX+x) = (char)(color->R*255);
      *(SaveG+y*vp->SizeX+x) = (char)(color->G*255);
      *(SaveB+y*vp->SizeX+x) = (char)(color->B*255);
   }
#ifdef SUN_DISPLAY
   pw_rop( SUN_PixWin, x, y, vp->ResX, vp->ResY, PIX_SRC | PIX_COLOR(col),
           NULL, 0, 0 );
#endif
   return;
}


/*
  ==============================================================================
   SaveImage --- save ray traced image in a file
  ==============================================================================
*/

BOOLEAN SaveImage ( vp )
VIEWPORT *vp;
{
```



```
   FILE *OutFile;
   int i;

   if ( !(OutFile = fopen( SaveFile, "w" ) ) )
      return FALSE;

   for ( i=0 ; i < vp->SizeX*vp->SizeY ; i++ )
   {
      putc( *(SaveR+i), OutFile );
      putc( *(SaveG+i), OutFile );
      putc( *(SaveB+i), OutFile );
   }
   fclose( OutFile );
   return TRUE;
}


/*
  ==============================================================================
   WriteLaser --- write picture to a postscript laser file (SUN specific)
  ==============================================================================
*/

BOOLEAN WriteLaser ( vp )
VIEWPORT *vp;
{
   FILE *OutFile;
   char pcent = '%';
   unsigned char pixval;
   int dx, dy, i, j;
   double sx, sy, scale, scaledX, scaledY;

   if ( ! (OutFile = fopen( DumpFile, "w" )) )
      return FALSE;

   sx = 2320 / (double)(vp->SizeX);
   sy = 3314 / (double)(vp->SizeY);
   scale = (sx > sy) ? sy : sx;
   scaledX = vp->SizeX*scale;
   scaledY = vp->SizeY*scale;
   dx = 72 + (2320 - scaledX)/2;
   dy = 72 + (3314 - scaledY)/2;

   fprintf( OutFile, "%c!\n", pcent );
   fprintf( OutFile, "gsave\n" );
   fprintf( OutFile, "initgraphics\n" );
   fprintf( OutFile, "0.24 0.24 scale\n" );
   fprintf( OutFile, "/imline %d string def\n", vp->SizeY );
   fprintf( OutFile, "/drawimage {\n" );
   fprintf( OutFile, "    %d %d 8\n", vp->SizeX, vp->SizeY );
   fprintf( OutFile, "    [%d 0 0 %d 0 %d]\n",vp->SizeX,1*vp->SizeY,vp->SizeY);
   fprintf( OutFile, "    { currentfile imline readhexstring pop } image\n" );
   fprintf( OutFile, "} def\n" );
   fprintf( OutFile, "%d %d translate\n", dx, dy );
   fprintf( OutFile, "%d %d scale\n",(int)(scaledX+0.5),-1*(int)(scaledY+0.5));
   fprintf( OutFile, "drawimage\n" );

   for ( i = 0 ; i < vp->SizeY ; i++ )
   {
      for ( j = 0 ; j < vp->SizeX ; j++ )
      {
#ifdef SUN_DISPLAY
         pixval = (unsigned char) pw_get( SUN_PixWin, i, j );
#endif
```

```
            fprintf( OutFile, "%02x", pixval );
         }
         fprintf( OutFile, "\n" );
      }

   fprintf( OutFile, "showpage\n" );
   fprintf( OutFile, "grestore\n" );

   fclose( OutFile );
   return TRUE;
}


/*
  ==========================================================================
   strequ --- test if two given strings are identical
  ==========================================================================
*/

BOOLEAN strequ( s1, s2 )
char *s1, *s2;
{
   while( *s1 == *s2++ )
      if ( *s1++ == '\0' ) return TRUE;
   return FALSE;
}


/*
  ==========================================================================
   GetWorld --- get data for world structure from file
  ==========================================================================
*/

void GetWorld( f, w )
FILE   *f;
WORLD *w;
{
   char sh[ MAX_NB_CHAR ];

   fscanf( f, "BackGnd = (%lf,%lf,%lf)",
           &(w->BackGnd.R), &(w->BackGnd.G), &(w->BackGnd.B) );
   fgetc( f );
   fscanf( f, "Shadows = %s", sh ); fgetc(f);
   w->Shadows = strequ( sh, "ON" );
   return;
}


/*
  ==========================================================================
   GetCamera --- get data for camera structure from file
  ==========================================================================
*/

void GetCamera( f, c )
FILE   *f;
CAMERA *c;
{
   fscanf( f, "Focal = %lf", &(c->Focal) );      fgetc( f );
   fscanf( f, "Distance = %lf", &(c->Dist) );    fgetc( f );
   fscanf( f, "ViewField = %lf", &(c->VField) ); fgetc( f );
   fscanf( f, "Rotation = (%lf,%lf,%lf)", &(c->Balance),
           &(c->LeftRight), &(c->TopBottom) );  fgetc( f );
```



```
   fscanf( f, "Center = (%lf,%lf,%lf)", &(c->Center.X),
           &(c->Center.Y), &(c->Center.Z) );    fgetc( f );
   return;
}


/*
  ==========================================================================
   GetSphere --- get data for sphere structure from file
  ==========================================================================
*/

void GetSphere( f, o )
FILE   *f;
OBJECT *o;
{
   SPHERE *s;

   o->IntersRoutine = IntSphere;
   o->NormalRoutine = NormSphere;
   s = (SPHERE *) malloc( sizeof(SPHERE) );
   o->Param = (void *) s;
   fscanf( f, "Center = (%lf,%lf,%lf)", &(s->C.X), &(s->C.Y), &(s->C.Z) );
   fgetc(f);
   fscanf( f, "Radius^2 = %lf", &(s->R2) ); fgetc(f);
   return;
}


/*
  ==========================================================================
   GetPlane --- get data for plane structure from file
  ==========================================================================
*/

void GetPlane( f, o )
FILE   *f;
OBJECT *o;
{
   PLANE *p;

   o->IntersRoutine = IntPlane;
   o->NormalRoutine = NormPlane;
   p = (PLANE *) malloc( sizeof(PLANE) );
   o->Param = (void *) p;
   fscanf( f, "Origin = (%lf,%lf,%lf)", &(p->Org.X),
           &(p->Org.Y), &(p->Org.Z) ); fgetc(f);
   fscanf( f, "Normal = (%lf,%lf,%lf)", &(p->Normal.X),
           &(p->Normal.Y), &(p->Normal.Z) ); fgetc(f);
   NORMALIZE( p->Normal );
   p->d = - DOT( p->Org, p->Normal );
   return;
}


/*
  ==========================================================================
   GetAlgSurf --- get data for algebraic surface structure from file
  ==========================================================================
*/

void GetAlgSurf( f, o )
FILE   *f;
OBJECT *o;
```

```
{
   o->IntersRoutine = IntAlgSurf;
   o->NormalRoutine = NormAlgSurf;
   o->Param = NULL;
   return;
}


/*
  ===========================================================================
   CalculatePolygon --- calculate data of polygon structure from data given
  ===========================================================================
*/

void CalculatePolygon( pol, a, b, c, d )
POLYGON *pol;
POINT   *a, *b, *c, *d;
{
   PLANE *p;

   MOVE( pol->a, *a );      MOVE( pol->d, *d );
   SUB( pol->ab, *b, *a ); SUB( pol->ac, *c, *a );
   SUB( pol->db, *b, *d ); SUB( pol->dc, *c, *d );
   p = &(pol->Pl);
   MOVE( p->Org, *a );
   PROD( p->Normal, pol->ab, pol->ac );
   NORMALIZE( p->Normal );
   p->d = - DOT( p->Org, p->Normal );
   return;
}


/*
  ===========================================================================
   GetPolygon --- get data for polygon structure from file
  ===========================================================================
*/

void GetPolygon( f, o )
FILE   *f;
OBJECT *o;
{
   POLYGON *pol;
   POINT    a, b, c, d;

   o->IntersRoutine = IntPolygon;
   o->NormalRoutine = NormPolygon;
   pol = (POLYGON *) malloc( sizeof(POLYGON) );
   o->Param = (void *) pol;
   fscanf( f, "a = (%lf,%lf,%lf)", &(a.X), &(a.Y), &(a.Z) ); fgetc(f);
   fscanf( f, "b = (%lf,%lf,%lf)", &(b.X), &(b.Y), &(b.Z) ); fgetc(f);
   fscanf( f, "c = (%lf,%lf,%lf)", &(c.X), &(c.Y), &(c.Z) ); fgetc(f);
   fscanf( f, "d = (%lf,%lf,%lf)", &(d.X), &(d.Y), &(d.Z) ); fgetc(f);
   CalculatePolygon( pol, &a, &b, &c, &d );
   return;
}


/*
  ===========================================================================
   CalcCoeff --- calculate coefficients of patch from control points
  ===========================================================================
*/
```



```
static double w[4][4] =
{
   {  1.0,  0.0,  0.0,  0.0 },
   { -3.0,  3.0,  0.0,  0.0 },
   {  3.0, -6.0,  3.0,  0.0 },
   { -1.0,  3.0, -3.0,  1.0 }
};

void CalcCoeff ( B, P )
double B[4][4][3], P[4][4][3];
{
   int i, j , k, l, m;
   double x;

   for ( k=0 ; k<4 ; k++ )
      for ( l=0 ; l<4 ; l++ )
      {
         P[l][k][0] = P[l][k][1] = P[l][k][2] = 0.0;
         for ( i=0 ; i<4 ; i++ )
            for ( j=0 ; j<4 ; j++ )
            {
               x = w[k][i]*w[l][j];
               for ( m = 0 ; m < 3 ; m++ ) P[l][k][m] += x*B[i][j][m];
            }
      }

   return;
}


/*
  ===========================================================================
   GetPatch --- get data for patch structure from file
  ===========================================================================
*/

void GetPatch( f, o )
FILE   *f;
OBJECT *o;
{
   double *p;
   SPLINE  B;
   FILE   *pf;
   char    fname[MAX_NB_CHAR];
   int     i, j;

   o->IntersRoutine = IntSpline;
   o->NormalRoutine = NormSpline;
   p = (double *) malloc( sizeof(SPLINE) );
   o->Param = (void *) p;
   fscanf( f, "Control points in %s", fname ); fgetc(f);
   pf = fopen( fname, "r" );
   for ( i=0 ; i<4 ; i++ )
   {
      for ( j=0 ; j<4 ; j++ )
      {
         fscanf( pf, "%lf %lf %lf", &(B[i][j][0]),
                                    &(B[i][j][1]), &(B[i][j][2]) );
         fgetc(pf);
      }
   }
   fclose(pf);
   CalcCoeff( B, p );
   return;
```

```
}


/*
  ==============================================================================
   GetBoundBox --- get data for bounding box structure from file
  ==============================================================================
*/

void GetBoundBox( f, o )
FILE *f;
OBJECT *o;
{
   BOX *b;

   o->BoundRoutine = IntBoundBox;
   b = (BOX *) malloc( sizeof(BOX ) );
   o->Bound = (void *) b;
   fscanf( f, "[%lf,%lf]x[%lf,%lf]x[%lf,%lf]", &(b->Xmin), &(b->Xmax),
           &(b->Ymin), &(b->Ymax), &(b->Zmin), &(b->Zmax) );
   fgetc(f);
   return;
}


/*
  ==============================================================================
   GetObject --- get data for object structure from file
  ==============================================================================
*/

void GetObject ( f, o )
FILE   *f;
OBJECT *o;
{
   ILLUM *i = &(o->Ill);
   char ObjName[MAX_NB_CHAR], IllumName[MAX_NB_CHAR], BoundName[MAX_NB_CHAR];

   fscanf( f, "Type = %s", ObjName ); fgetc(f);

   if ( strequ(ObjName,"SPHERE") ) GetSphere( f, o );
   if ( strequ(ObjName,"PLANE" ) ) GetPlane ( f, o );
   if ( strequ(ObjName,"POLYGON") ) GetPolygon( f, o );
   if ( strequ(ObjName,"ALGEBRAIC_SURFACE") ) GetAlgSurf( f, o );
   if ( strequ(ObjName,"BEZIER_PATCH") ) GetPatch( f, o );

   fscanf( f, "Illumination = %s", IllumName ); fgetc(f);
   if ( strequ(IllumName,"BLACKBODY" ) ) o->IllumId = BLACKBODY;
   if ( strequ(IllumName,"AMBIANT"    ) ) o->IllumId = AMBIANT;
   if ( strequ(IllumName,"DIFFUSE"    ) ) o->IllumId = DIFFUSE;
   if ( strequ(IllumName,"TEXTURE"    ) ) o->IllumId = TEXTURE;
   if ( strequ(IllumName,"SPECULAR"   ) ) o->IllumId = SPECULAR;
   if ( strequ(IllumName,"MIRROR"     ) ) o->IllumId = MIRROR;
   if ( strequ(IllumName,"TRANSPARENT")) o->IllumId = TRANSPARENT;
   fscanf( f, "Cs = (%lf,%lf,%lf)", &(i->Cs.R), &(i->Cs.G), &(i->Cs.B) );
   fgetc(f);
   fscanf( f, "Cr = (%lf,%lf,%lf)", &(i->Cr.R), &(i->Cr.G), &(i->Cr.B) );
   fgetc(f);
   fscanf( f, "Ka = %lf Kd = %lf Ks = %lf Kt = %lf",
          &(i->Ka), &(i->Kd), &(i->Ks), &(i->Kt) ); fgetc(f);
   fscanf( f, "n = %lf Kn = %lf", &(i->n), &(i->Kn) ); fgetc(f);
   fscanf( f, "Bounding volume = %s", BoundName ); fgetc(f);
   if ( strequ(BoundName,"NONE") ) o->Bound = NULL;
   if ( strequ(BoundName,"BOX" ) ) GetBoundBox( f, o );
```



```
   return;
}


/*
  ==============================================================================
   GetLight --- get data for light structure from file
  ==============================================================================
*/

void GetLight ( f, l )
FILE  *f;
LIGHT *l;
{
   fscanf( f, "Source = (%lf,%lf,%lf)", &(l->Source.X),
           &(l->Source.Y), &(l->Source.Z) ); getc(f);
   fscanf( f, "Color = (%lf,%lf,%lf)", &(l->Col.R), &(l->Col.G), &(l->Col.B) );
   fgetc(f);
   return;
}


/*
  ==============================================================================
   GetViewPort --- get data for viewport structure from file
  ==============================================================================
*/

void GetViewPort ( f, vp )
FILE     *f;
VIEWPORT *vp;
{
   fscanf( f, "Size = (%d,%d)", &(vp->SizeX), &(vp->SizeY) ); fgetc(f);
   fscanf( f, "Origin = (%d,%d)", &(vp->OrgX), &(vp->OrgY) ); fgetc(f);
   fscanf( f, "Resolution = (%d,%d)", &(vp->ResX), &(vp->ResY) ); fgetc(f);
   return;
}


/*
  ==============================================================================
   GetTexture --- get texture image from file
  ==============================================================================
*/

void GetTexture ( f )
FILE *f;
{
   char fn[MAX_NB_CHAR];
   FILE *ff;
   int i;

   fscanf( f, "Size = %dx%d", &SizeTextureX, &SizeTextureY ); fgetc(f);
   SizeTexture = SizeTextureX*SizeTextureY;
   fscanf( f, "File Name = %s", fn ); fgetc(f);
   ff = fopen( fn, "r" );
   TextureR = (unsigned char *) malloc( SizeTexture );
   TextureG = (unsigned char *) malloc( SizeTexture );
   TextureB = (unsigned char *) malloc( SizeTexture );
   for ( i=0 ; i<SizeTexture ; i++ ) *(TextureR+i) = fgetc(ff);
   for ( i=0 ; i<SizeTexture ; i++ ) *(TextureG+i) = fgetc(ff);
   for ( i=0 ; i<SizeTexture ; i++ ) *(TextureB+i) = fgetc(ff);
   fclose( ff );
   fscanf( f, "Unit = %lf", &UnitTexture ); fgetc(f);
```

```
    return;
}


/*
   ==========================================================================
    GetSave --- get save data file name and set save flag
   ==========================================================================
*/

void GetSave ( f )
FILE *f;
{
    SaveFlag = TRUE;
    fscanf( f, "File Name = %s", SaveFile ); fgetc(f);
    return;
}


/*
   ==========================================================================
    GetDump --- get dump post script file name and set dump flag
   ==========================================================================
*/

void GetDump ( f )
FILE *f;
{
    DumpFlag = TRUE;
    fscanf( f, "File Name = %s", DumpFile ); fgetc(f);
    return;
}


/*
   ==========================================================================
    InitScene --- initialises structures describing a scene
   ==========================================================================
*/

BOOLEAN InitScene( fname, w, camera, viewport )
char     *fname;
WORLD    *w;
CAMERA   *camera;
VIEWPORT *viewport;
{
    FILE   *f;
    char   buffer[ MAX_NB_CHAR ];
    char   com[ MAX_NB_CHAR ];
    short  no = 0, nl = 0;

    SaveFlag = FALSE;
    DumpFlag = FALSE;
    if ( ! (f = fopen( fname, "r" )) ) return FALSE;

    while ( fgets( buffer, MAX_NB_CHAR, f ) != NULL )
    {
        if ( sscanf( buffer, "#%s", com ) != 1 ) continue;
        if ( strequ(com,"world"    ) ) GetWorld    ( f, w );
        if ( strequ(com,"camera"   ) ) GetCamera   ( f, camera );
        if ( strequ(com,"viewport") ) GetViewPort( f, viewport );
        if ( strequ(com,"object"   ) ) GetObject   ( f, &(w->Objects[no++]) );
        if ( strequ(com,"light"    ) ) GetLight    ( f, &(w->Lights[nl++]) );
        if ( strequ(com,"texture"  ) ) GetTexture  ( f );
```



```
        if ( strequ(com,"save"     ) ) GetSave     ( f );
        if ( strequ(com,"dump"     ) ) GetDump     ( f );
    }

    w->NbObjects = no; w->NbLights = nl;
    fclose( f );
    if ( SaveFlag )
    {
    if (!(SaveR=(char *)malloc(viewport->SizeX*viewport->SizeY))) return FALSE;
    if (!(SaveG=(char *)malloc(viewport->SizeX*viewport->SizeY))) return FALSE;
    if (!(SaveB=(char *)malloc(viewport->SizeX*viewport->SizeY))) return FALSE;
    }

    return TRUE;
}


/*
   ==========================================================================
    CleanUp --- deallocate all ressources used by the program (SUN specific)
   ==========================================================================
*/

void CleanUp( vp )
VIEWPORT *vp;
{
    if ( SaveFlag )
    {
        if ( ! SaveImage( vp ) )
            puts( "cannot save image" );
    }
    if ( DumpFlag )
    {
        if ( ! WriteLaser( vp ) )
            puts( "cannot create post script file" );
    }
#ifdef SUN_DISPLAY
    window_set( SUN_Frame, FRAME_NO_CONFIRM, TRUE, 0 );
    window_destroy( SUN_Frame );
#endif
    return;
}
```

```
/*
  ==========================================================================
  RAY TRACER --- ray2.c
  --------------------------------------------------------------------------
  Content       : Intersection logic for algebraic surfaces.
  Author        : Stam Jos
  Creation Date : 01/03/88
  Last Modified : 22/07/88
  ==========================================================================
*/

#include "ray.h"


double sqrt();
#define MAX_POL_DEG 10


/**************************** GLOBAL VARIABLES DEFINITION ******************/

extern STURM_SERIE S[MAX_POL_DEG];
extern int NbSturm;


/*************************** MODULES DEFINITION ***************************/

/*
  ==========================================================================
  Polynomial --- calculate polynomial corresponding to the ray
  ==========================================================================
*/

void Polynomial( ray )
RAY *ray;
{
    double u1, u2, u3, u4, u5, u6, u7, u8;
    STURM_SERIE *S0 = &(S[0]);

    u1 = ray->Org.X*ray->Org.X; u2 = ray->Org.Y*ray->Org.Y;
    u3 = ray->Dir.X*ray->Dir.X; u4 = ray->Dir.Y*ray->Dir.Y;
    u5 = ray->Org.X*ray->Dir.X; u6 = ray->Org.Y*ray->Dir.Y;
    u7 = u3+u4; u8 = u1+u2;

    S0->a[0] = u7*u7;
    S0->a[1] = 4.0*u7*(u5+u6);
    S0->a[2] = 6.0*(u5*u5+u6*u6) - 10.0*u7 + ray->Dir.Z*ray->Dir.Z*15.0 +
               2.0*(u1*u4+4.0*u5*u6+u3*u2);
    S0->a[3] = 2.0*( (2.0*u8-10.0)*(u5+u6) + ray->Org.Z*ray->Dir.Z*15.0);
    S0->a[4] = u8*(u8-10.0) + ray->Org.Z*ray->Org.Z*15.0 + 9.0;

    S0->deg = 4;

    return;
}


/*
  ===================================================================
  Sturm --- compute Sturm serie for given polynomial
  ===================================================================
*/

int Sturm( deg )
int deg;
```

```
{
   int          i, j, n0 = deg;
   double       a0, b0, c0, x;
   STURM_SERIE *S0, *S1, *S2;

   S0 = &(S[0]); S1 = &(S[1]);
   S0->deg = n0; S1->deg = n0-1;
   for ( i = 0 ; i <  n0 ; i++ ) S1->a[i] = (n0-i)*S0->a[i];
   S1->a[n0] = 0.0;
   for ( j = 2 ; j <= n0 ; j++ )
   {
      a0 = S0->a[0]; b0 = S1->a[0];
      c0 = b0*S0->a[1] - a0*S1->a[1];
      S2 = &(S[j]); x = 1.0;
      for ( i = 1 ; i <= n0-j+1 ; i++ )
      {
         S2->a[i-1] = (c0*S1->a[i] - b0*(b0*S0->a[i+1]-a0*S1->a[i+1]))/x;
         if ( i == 1 ) { x=S2->a[0]; if ( x < 0 ) x = -x; }
      }
      S2->a[n0-j+1] = 0.0; S2->a[0] /= x;
      S2->deg = n0-j;
      S0 = S1; S1 = S2;
      while ( S1->a[0] == 0.0 )
      {
         for ( i = 0 ; i <= n0-j ; i++ ) S1->a[i] = S1->a[i+1];
         S1->a[i] = 0.0;
         if ( --n0 <= j ) return n0;
      }
   }
   return n0;
}


/*
  ======================================================================
   V --- evaluate a Sturm Polynomial by Horners method
  ======================================================================
*/

double V( i, x )
int i;
double x;
{
   short j;
   STURM_SERIE *s = &(S[i]);
   double v = s->a[0];

   for ( j=1 ; j<= s->deg ; j++ ) v = v*x + s->a[j];

   return v;
}


/*
  ======================================================================
   W --- evaluate # sign variation in the Sturm serie for given value
  ======================================================================
*/

int W( x )
double x;
{
   short i, nb = 0;
   double f = V(0,x), g;
```



```
   for ( i=1 ; i<=NbSturm ; i++ )
   {
      if ( (f * (g = V(i,x))) < 0.0 ) nb++;
      f = g;
   }
   return nb;
}


/*
  ============================================================================
   IntAlgSurf --- calculate intersection of ray with algebraic surface
  ============================================================================
*/

BOOLEAN IntAlgSurf( ray, nil, patch, objnum, shadow, t1, t2 )
RAY     *ray;
void    *nil;
PATCH   *patch;
short    objnum;
BOOLEAN shadow;
double   t1, t2;
{
   int n1, n2, n;
   double m;

   Polynomial( ray );
   NbSturm = Sturm( S[0].deg );

   if ( (n1=W(t1)) - (n2=W(t2)) == 0 ) return FALSE;
   while ( n1 - n2 != 1 )
   {
      m = (t1+t2) / 2.0;
      if ( n1 - (n=W(m)) == 0.0 ) { t1 = m; n1 = n; }
      else                        { t2 = m; n2 = n; }
   }

   while ( t2 - t1 > PRECISION )
   {
      m = (t1+t2) / 2.0;
      if ( V(0,t1)*V(0,m) < 0.0 ) t2 = m;
      else                        t1 = m;
   }

   if ( t2 >= patch->Dist ) return TRUE;
   patch->Dist = t2;
   if ( shadow ) return TRUE;
   patch->ObjNum = objnum;
   GETPOINT( patch->Inters, ray->Org, t2, ray->Dir );

   return TRUE;
}


/*
  ============================================================================
   NormAlgSurf --- calculate normal vector for a given point on alg. surf.
  ============================================================================
*/

void NormAlgSurf ( nil, patch )
void  *nil;
PATCH *patch;
```

```
{
   double x = patch->Inters.X,
          y = patch->Inters.Y,
          z = patch->Inters.Z;

   patch->Normal.X = 4.0*x*(x*x+y*y-5.0);
   patch->Normal.Y = 4.0*y*(x*x+y*y-5.0);
   patch->Normal.Z = 2.0*z*15.0;

   NORMALIZE( patch->Normal );
   return;
}
```



```
/*
  ===============================================================================
   RAY TRACER --- ray3.c
  -------------------------------------------------------------------------------
   Content       : Intersection logic for algebraic parametric surfaces
   Author        : Stam Jos
   Creation Date : 18/07/88
   Last Modified : 25/07/88
  ===============================================================================
*/

#include "ray.h"


/************************ GLOBAL VARIABLES DEFINITION **********************/

static int Pu[NBPOL][NBMON],    /* powers of u for each monomial */
           Pv[NBPOL][NBMON];    /* powers of v for each monomial */
static int Long[NBPOL];         /* number of monomials for each polynomial */
static double F[NBPOL][NBMON];  /* value of coefficients for each monomial */
static int NbBase;              /* number of polynomials in the basis */

#define TEMP1 (NBPOL-1) /* index of first scratch polynomial */
#define TEMP2 (NBPOL-2) /* index of second scratch polynomial */

STURM_SERIE S[MAX_DEG]; /* Sturm series for polynomial root finding */
int NbSturm;            /* number of Sturm polynomials */

static INTERVAL Inter[MAX_DEG]; /* sequence of root isolating intervals */
static double Root[MAX_DEG];    /* roots contained in the intervals */
static double RootV[MAX_DEG];   /* v-coordinates of a root */
static int NbInt;               /* number of intervals */
static int NbRoots;             /* number of roots */

static int vi;                  /* non-zero coordinate of direction vector */
static double Org[3], Dir[3];   /* origin and direction of a ray */

static double Uint, Vint;       /* (u,v) of intersection point */

int GroebnerSize[MAX_DEG], DegPolU[MAX_DEG], DegPolV[MAX_DEG], NbCrit;

/********************** EXTERNAL MODULES DEFINITION ************************/

extern int    Sturm(), W();
extern double V();


/*************************** MODULES DEFINITION ***************************/

/*
  ===============================================================================
   Greater --- tests if monomial u^i1v^i2 is greater then u^j1v^j2, in the
               sense of the Total Lexicographical Order.
  ===============================================================================
*/

BOOLEAN Greater ( i1, i2, j1, j2 )
int i1, i2, j1, j2;
{
   if ( i2 > j2 ) return TRUE;
   return ( i2 == j2 && i1 > j1 );
}
```

```
/*
  ================================================================================
   Sum --- computes the sum of polynomials: Fi3 = Fi1 + Fi2
  ================================================================================
*/

void Sum ( i1, i2, i3 )
int i1, i2, i3;
{
   int j1, j2, j3;      /* index for monomials */
   int u1, u2, v1, v2;  /* powers of u and v */
   int l1, l2;          /* length of Fi1 and Fi2 */
   int u3, v3;          /* scratch use */
   double F3;           /* scratch use */

   j1 = j2 = j3 = 0;
   l1 = Long[i1]; l2 = Long[i2];

   while ( j1 < l1 || j2 < l2 )
   {
      if ( j1 == l1 ) u1 = v1 = -1; else { u1 = Pu[i1][j1]; v1 = Pv[i1][j1]; }
      if ( j2 == l2 ) u2 = v2 = -1; else { u2 = Pu[i2][j2]; v2 = Pv[i2][j2]; }

      u3 = u1; v3 = v1;
      if      ( u1 == u2 && v1 == v2 ) F3 = F[i1][j1++] + F[i2][j2++];
      else if ( Greater(u1,v1,u2,v2) ) F3 = F[i1][j1++];
      else                             { F3 = F[i2][j2++]; u3 = u2; v3 = v2; }

      if ( ! ZERO(F3) )
      {
         F[i3][j3] = F3; Pu[i3][j3] = u3; Pv[i3][j3++] = v3;
      }
   }
   Long[i3] = j3;
   return;
}


/*
  ================================================================================
   Scale --- divide polynomial Fi by the coefficient of LM(Fi)
  ================================================================================
*/

void Scale ( i )
int i;
{
   double s; /* scaling factor */
   int j;    /* index for monomials */

   s = F[i][0];
   if ( ZERO(1.0-s) ) return;
   s = 1.0 / s;
   F[i][0] = 1.0;
   for ( j=1 ; j < Long[i] ; j++ ) F[i][j] *= s;
   return;
}


/*
  ================================================================================
   Reduce --- try to reduce monomial Fi1.j1 with monomial LM(Fi2), returns TRUE
              if a reduction occured.
  ================================================================================
```



```
*/

BOOLEAN Reduce ( i1, j1, i2 )
int i1, j1, i2;
{
   int du, dv; /* difference in powers between two monomials */
   int j;      /* index of monomial */
   double F0;  /* value of coefficient of monomial Fi1.j1 */

   if ( Long[i1] == 0 || Long[i2] == 0 ) return FALSE;
   if ( (du = Pu[i1][j1] - Pu[i2][0]) < 0 ) return FALSE;
   if ( (dv = Pv[i1][j1] - Pv[i2][0]) < 0 ) return FALSE;
   F0 = -F[i1][j1];

   for ( j=0 ; j<Long[i1] ; j++ )
   {
      Pu[TEMP1][j] = Pu[i1][j]; Pv[TEMP1][j] = Pv[i1][j];
      F[TEMP1][j] = F[i1][j];
   }
   for ( j=0 ; j<Long[i2] ; j++ )
   {
      Pu[TEMP2][j] = Pu[i2][j] + du; Pv[TEMP2][j] = Pv[i2][j] + dv;
      F[TEMP2][j] = F0*F[i2][j];
   }
   Long[TEMP1] = Long[i1]; Long[TEMP2] = Long[i2];

   Sum( TEMP1, TEMP2, i1 );
   return TRUE;
}


/*
  ================================================================================
   S_Polynomial --- calculates "S-polynomial" of Fi1 and Fi2 and puts the
                    result in Fi3
  ================================================================================
*/

void S_Polynomial ( i1, i2, i3 )
int i1, i2, i3;
{
   int u1, v1; /* amount by which powers of u and v of Fi1 must be incr. */
   int u2, v2; /* same for Fi2 */
   int du, dv; /* max of these powers */
   int j;      /* index of a monomial */

   if ( Long[i1] == 0 || Long[i2] == 0 ) { Long[i3] = 0; return; }
   du = MAX(Pu[i1][0],Pu[i2][0]); dv = MAX(Pv[i1][0],Pv[i2][0]);
   u1 = du - Pu[i1][0]; v1 = dv - Pv[i1][0];
   u2 = du - Pu[i2][0]; v2 = dv - Pv[i2][0];

   Long[TEMP1] = Long[i1] - 1;
   for ( j=0 ; j<Long[TEMP1] ; j++ )
   {
      Pu[TEMP1][j] = Pu[i1][j+1] + u1; Pv[TEMP1][j] = Pv[i1][j+1] + v1;
      F[TEMP1][j] = F[i1][j+1];
   }
   Long[TEMP2] = Long[i2] - 1;
   for ( j=0 ; j<Long[TEMP2]; j++ )
   {
      Pu[TEMP2][j] = Pu[i2][j+1] + u2; Pv[TEMP2][j] = Pv[i2][j+1] + v2;
      F[TEMP2][j] = -F[i2][j+1];
   }
```

```
   Sum( TEMP1, TEMP2, i3 );
   Scale( i3 );
   return;
}


/*
  ==============================================================================
   Reduce_2 --- reduce the two polynomials Fi1 and Fi2 mutually.
  ==============================================================================
*/

void Reduce_2 ( i1, i2 )
int i1, i2;
{
   int i0;

   if ( Pu[i1][0] < Pu[i2][0] ) { i0=i1; i1=i2; i2=i0; }
   while ( Reduce( i1, 0, i2, 0) )
   {
      if ( Long[i1] == 0 ) return;
      Scale( i1 ); Scale( i2 );
      if ( Pu[i1][0] < Pu[i2][0] ) { i0=i1; i1=i2; i2=i0; }
   }
   return;
}


/*
  ==============================================================================
   Exchange --- exchange polynomials Fi and Fj
  ==============================================================================
*/

void Exchange( i, j )
int i, j;
{
   int k, l;

   for ( k=0 ; k<Long[i] ; k++ )
   {
      Pu[TEMP1][k] = Pu[i][k]; Pv[TEMP1][k] = Pv[i][k]; F[TEMP1][k] = F[i][k];
   }
   for ( k=0 ; k<Long[j] ; k++ )
   {
      Pu[i][k] = Pu[j][k]; Pv[i][k] = Pv[j][k]; F[i][k] = F[j][k];
   }
   for ( k=0 ; k<Long[i] ; k++ )
   {
      Pu[j][k] = Pu[TEMP1][k]; Pv[j][k] = Pv[TEMP1][k]; F[j][k] = F[TEMP1][k];
   }
   l = Long[i]; Long[i] = Long[j]; Long[j] = l;
   return;
}


/*
  ==============================================================================
   Groebner --- computes Groebner basis corresponding to the 2 polynomials,
                returns index of univariate polynomial in K[u];
  ==============================================================================
*/

int Groebner ()
```



```
{
   int i;

   i = 0;
   Reduce_2( 0, 1 );
   if ( Greater( Pu[1][0], Pv[1][0], Pu[0][0], Pv[0][0] ) ) Exchange(0,1);
   while ( Long[i+1] != 0 )
   {
      S_Polynomial( i, i+1, i+2 );
      Reduce_2( i+2, i );
      if ( Greater(Pu[i+2][0],Pv[i+2][0],Pu[i][0],Pv[i][0]) ) Exchange(i+2,i);
      i++;
      Reduce_2( i+1, i );
      if ( Greater(Pu[i+1][0],Pv[i+1][0],Pu[i][0],Pv[i][0]) ) Exchange(i+1,i);
   }
   return (i);
}


/*
  ==============================================================================
   Zero --- find root contained in [Umin,Umax]
  ==============================================================================
*/

double Zero( Umin, Umax )
double Umin, Umax;
{
   double m, Vm;

   while ( Umax - Umin > SMALL || !ZERO(Vm) )
   {
      m = (Umax+Umin) / 2.0;
      Vm = V(0,m);
      if ( V(0,Umin)*Vm < 0.0 ) Umax = m;
      else Umin = m;
   }

   return m;
}


/*
  ==============================================================================
   Isolate --- isolate all roots in [Umin,Umax] with intervals
  ==============================================================================
*/

void Isolate ( Umin, Umax )
double Umin, Umax;
{
   double m;
   int    nb;

   if ( (nb = W(Umin) - W(Umax)) == 0 ) return;
   if ( nb == 1 )
   {
      Inter[NbInt].a = Umin;
      Inter[NbInt].b = Umax;
      NbInt++;
      return;
   }
   m = (Umin+Umax) / 2.0;
```

```
   Isolate( Umin, m );
   Isolate( m, Umax );
   return;
}


/*
  ==============================================================================
   Roots --- calculates roots of univariate polynomial Fi
  ==============================================================================
*/

void Roots ( i )
int i;
{
   int j, k, deg;
   double p, q, delta, sqrt();

   deg = Pu[i][0];
   for ( j=0 ; j < Long[i] ; j++ ) S[0].a[deg-Pu[i][j]] = F[i][j];
   while ( ZERO(S[0].a[deg]) )
   {
      for ( k = 0 ; k < deg ; k++ ) S[0].a[k] = S[0].a[k+1];
      deg--;
   }

   DegPolU[deg]++;
   NbRoots = 0;
   if ( deg == 1 )
   {
      Root[0] = -S[0].a[1] / S[0].a[0];
      if ( Root[0] < 0.0 || Root[0] > 1.0 ) return;
      NbRoots = 1;
      return;
   }
   else if ( deg == 2 )
   {
      p = S[0].a[1] / (2.0*S[0].a[0]); q = S[0].a[2] / S[0].a[0];
      if ( (delta=p*p-q) < 0.0 ) return;
      if ( delta == 0.0 )
      {
         if ( p > 0.0 || p < -1.0 ) return;
         NbRoots = 1; Root[0] = -p;
         return;
      }
      delta = sqrt(delta);
      Root[0] = -(p+delta); j = 1;
      if ( Root[0] < 0.0 || Root[0] > 1.0 ) j = 0;
      Root[j] = Root[0] + 2.0*delta;
      if ( Root[j] < 0.0 || Root[j] > 1.0 ) j--;
      NbRoots = j+1;
      return;
   }
   S[0].deg = deg;
   NbSturm = Sturm( deg );
   NbInt = 0;
   Isolate(0.0,1.0);
   NbRoots = NbInt;
   for ( j = 0 ; j < NbRoots ; j++ ) Root[j] = Zero(Inter[j].a,Inter[j].b);
   return;
}


/*
```



```
  ==============================================================================
   VPol --- calculates v-polynomial by substituting u0 in FN
  ==============================================================================
*/

void VPol ( u0, N )
double u0;
int N;
{
   double u, uu;
   int    i, j, k, j0;

   j0 = i = 0;
   for ( j = Pv[N][0] ; j >= 0 ; j-- )
   {
      if ( Pv[N][i] == j )
      {
         uu = u = F[N][i];
         for ( k = Pu[N][i++]-1 ; k >= 0 ; k-- )
         {
            u *= u0;
            if ( Pu[N][i] == k ) u += F[N][i++];
         }
         if ( j0 != 0 ) S[0].a[j0++] = u;
         else          { if ( !ZERO(u/uu) ) S[0].a[j0++] = u; }
      }
      else if ( j0 !=  0 ) S[0].a[j0++] = 0.0;
   }
   S[0].deg = j0-1;
   return;
}


/*
  ==============================================================================
   VRoots --- calculates v-coordinates of root with u-coordinate u0
  ==============================================================================
*/

void VRoots ( u0, N )
double u0;
int N;
{
   int N0 = N, j, deg;
   double p, q, delta, sqrt();

   do { VPol( u0, --N0 ); } while ( S[0].deg < 0 && N0 > 0 );

   deg = S[0].deg;
   while ( ZERO(S[0].a[deg]) )
   {
      for ( j = 0 ; j < deg ; j++ ) S[0].a[j] = S[0].a[j+1];
      deg--;
   }
   S[0].deg = deg;

   DegPolV[deg]++;
   NbInt = 0;
   if ( deg == 1 )
   {
      RootV[0] = -S[0].a[1] / S[0].a[0];
      if ( RootV[0] < 0.0 || RootV[0] > 1.0 ) return;
      NbInt = 1;
      return;
```

```
   }
   else if ( deg == 2 )
   {
      p = S[0].a[1] / (2.0*S[0].a[0]); q = S[0].a[2] / S[0].a[0];
      if ( (delta=p*p-q) < 0.0 ) return;
      if ( delta == 0.0 )
      {
         if ( p > 0.0 || p < -1.0 ) return;
         NbInt = 1; RootV[0] = -p;
         return;
      }
      delta = sqrt(delta);
      RootV[0] = -(p+delta); j = 1;
      if ( RootV[0] < 0.0 || RootV[0] > 1.0 ) j = 0;
      RootV[j] = RootV[0] + 2.0*delta;
      if ( RootV[j] < 0.0 || RootV[j] > 1.0 ) j--;
      NbInt = j+1;
      return;
   }
   NbSturm = Sturm( deg );
   NbInt = 0;
   Isolate(0.0,1.0);
   for ( j = 0 ; j < NbInt ; j++ ) RootV[j] = Zero(Inter[j].a,Inter[j].b);
   return;
}


/*
   ============================================================================
   CalcBasis --- calculate the two polynomials of the ideal basis
   ============================================================================
*/

void CalcBasis ( P )
SPLINE  P;
{
   int     i, j, k, l, v1, v2, v3;
   double x, a00, b00;

   i = j = 0;
   if        ( !ZERO(Dir[0]) ) { v1 = 0; v2 = 1; v3 = 2; vi = 0; }
   else if ( !ZERO(Dir[1]) ) { v1 = 1; v2 = 0; v3 = 2; vi = 1; }
   else                        { v1 = 2; v2 = 0; v3 = 1; vi = 2; }

   for ( k=3 ; k>=0 ; k-- )
      for ( l=3 ; l>=0 ; l-- )
      {
         x = Dir[v2]*P[l][k][v1] - Dir[v1]*P[l][k][v2];
         if ( !ZERO(x) ) { Pu[0][i] = l; Pv[0][i] = k; F[0][i++] = x; }
         x = Dir[v3]*P[l][k][v1] - Dir[v1]*P[l][k][v3];
         if ( !ZERO(x) ) { Pu[1][j] = l; Pv[1][j] = k; F[1][j++] = x; }
      }
   a00 = Dir[v1]*Org[v2] - Dir[v2]*Org[v1];
   b00 = Dir[v1]*Org[v3] - Dir[v3]*Org[v1];
   if ( Pu[0][i-1] == 0 && Pv[0][i-1] == 0 ) F[0][--i] += a00;
   else { Pu[0][i] = Pv[0][i] = 0; F[0][i] = a00; }
   if ( !ZERO(F[0][i]) ) i++;
   if ( Pu[1][j-1] == 0 && Pv[1][j-1] == 0 ) F[1][--j] += b00;
   else { Pu[1][j] = Pv[1][j] = 0; F[1][j] = b00; }
   if ( !ZERO(F[1][j]) ) j++;
   Long[0] = i; Long[1] = j;
   Scale(0); Scale(1);
   return;
}
```



```
/*
   ============================================================================
   CalcV --- calculate value of polynomial coefficient of u^i corresponding
             to v
   ============================================================================
*/

double CalcV ( P, v, i )
SPLINE  P;
double v;
int    i;
{
   int j;
   double u = P[i][3][vi];

   for ( j=2 ; j >= 0 ; j-- ) u = u*v + P[i][j][vi];

   return u;
}


/*
   ============================================================================
   CalcT --- calculate value of t parameter corresponding to (u,v)
   ============================================================================
*/

double CalcT ( P, u, v )
SPLINE P;
double u, v;
{
   int i;
   double t = CalcV( P, v, 3 );

   for ( i = 2 ; i >= 0 ; i-- ) t = t*u + CalcV( P, v, i );

   return ( (t-Org[vi]) / Dir[vi] );
}


/*
   ============================================================================
   IntSpline --- calculate intersection of ray with surface
   ============================================================================
*/

BOOLEAN IntSpline ( ray, P, patch, objnum, shadow, t1, t2 )
RAY      *ray;
SPLINE    P;
PATCH    *patch;
short     objnum;
BOOLEAN   shadow;
double    t1, t2;
{
   int     i, j, N;
   double t, t0, v, o;
   char    c;

   Org[0] = ray->Org.X; Org[1] = ray->Org.Y; Org[2] = ray->Org.Z;
   Dir[0] = ray->Dir.X; Dir[1] = ray->Dir.Y; Dir[2] = ray->Dir.Z;
   CalcBasis( P );
   N=Groebner();
```

```
    GroebnerSize[N]++;
    if ( Pu[N][0] == 0 && Pv[N][0] == 0 ) { NbCrit++; return FALSE; }

    Roots( N );
    t0 = t2;
    for ( i = 0 ; i < NbRoots ; i++ )
    {
       VRoots( Root[i], N );
       for ( j = 0 ; j < NbInt ; j++ )
       {
          t = CalcT( P, Root[i], RootV[j] );
          if ( t > t1 && t < t0 ) { Uint = Root[i]; Vint = RootV[j]; t0 = t; }
       }
    }
    if ( t0 == t2 || t0 >= patch->Dist ) return FALSE;

    patch->Dist = t0;
    if ( shadow ) return TRUE;
    patch->ObjNum = objnum;
    GETPOINT( patch->Inters, ray->Org, t0, ray->Dir );

    return TRUE;
}


/*
  ===============================================================================
   CalcDu --- calculate polynomial coefficient in v of u^i of dP/du
  ===============================================================================
*/

double CalcDu ( P, v, i, xyz )
SPLINE P;
double v;
int    i, xyz;
{
   return (P[i][0][xyz]+v*(P[i][1][xyz]+v*(P[i][2][xyz]+v*P[i][3][xyz])));
}


/*
  ===============================================================================
   CalcDv --- calculate polynomial coefficient in u of v^j of dP/dv
  ===============================================================================
*/

double CalcDv ( P, u, j, xyz )
SPLINE P;
double u;
int    j, xyz;
{
   return (P[0][j][xyz]+u*(P[1][j][xyz]+u*(P[2][j][xyz]+u*P[3][j][xyz])));
}


/*
  ===============================================================================
   CalcdPdu --- calculate 'xyz' coordinate of derivative vector in direction u
  ===============================================================================
*/

double CalcdPdu ( P, u, v, xyz )
SPLINE P;
double u, v;
```



```
int    xyz;
{
   return (CalcDu(P,v,1,xyz)+u*(2*CalcDu(P,v,2,xyz)+3*u*CalcDu(P,v,3,xyz)));
}


/*
  ===============================================================================
   CalcdPdv --- calculate 'xyz' coordinate of derivative vector in direction v
  ===============================================================================
*/

double CalcdPdv ( P, u, v, xyz )
SPLINE P;
double u, v;
int    xyz;
{
   return (CalcDv(P,u,1,xyz)+v*(2*CalcDv(P,u,2,xyz)+3*v*CalcDv(P,u,3,xyz)));
}


/*
  ===============================================================================
   NormSpline --- calculate normal vector for a given point on the surface
  ===============================================================================
*/

void NormSpline ( P, patch )
SPLINE  P;
PATCH  *patch;
{
   VECTOR dPdu, dPdv;

   dPdu.X = CalcdPdu(P,Uint,Vint,0);
   dPdu.Y = CalcdPdu(P,Uint,Vint,1);
   dPdu.Z = CalcdPdu(P,Uint,Vint,2);

   dPdv.X = CalcdPdv(P,Uint,Vint,0);
   dPdv.Y = CalcdPdv(P,Uint,Vint,1);
   dPdv.Z = CalcdPdv(P,Uint,Vint,2);

   PROD( patch->Normal, dPdu, dPdv );
   NORMALIZE( patch->Normal );

   return;
}
```

# Bibliographie


[1] D. Brunet, *Ray-Tracing sur PC*, mémoire de licence ès sciences informatiques, Université de Genève, 1987.

[2] G.E. Collins, et R. Loos, *Real Zeros of polynomials*, Computing Suppl. 4, pages 83-94, 1982.

[3] E. Durand, *Solutions Numérique des Equations Algébriques*, Tome I, Masson et Cie., pages 183-188, 1960.

[4] E. Graham, *Graphic Scene Simulations*, AMIGA World, No 11 Mai/Juin, pages 18-24, 1987.

[5] P. Hanrahan, *Ray Tracing Algebraic Surfaces*, Computer Graphics (Proc. SIGGRAPH 84), Vol. 17, No. 3, pages 83-90, 1983.

[6] J.T. Kajiya, *Ray Tracing Parametric Patches*, Computer Graphics (Proc. SIGGRAPH 82), Vol. 16, No. 3, pages 245-254, 1982.

[7] B.W. Kernighan, et D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[8] J.M. Ortega, et W.C. Reinbolt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[9] D. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, 1985.

[10] T.W. Sederberg, et D.C. Anderson, *Ray Tracing of Steiner Patches*, Computer Graphics (Proc. SIGGRAPH 84), Vol. 18, No.3, pages 159-164, 1984.

[11] J. Snyder, et A. Barr, *Ray Tracing Complex Models Containing Surface Tesselations*, Computer Graphics (Proc. SIGGRAPH 87), Vol. 21, No. 4, pages 119-128, 1987.

[12] M.A.J. Sweeney, et R.H. Bartels, *Ray Tracing Free-form B-spline Surfaces*, IEEE Computer Graphics and Applications 6(2), pages 41-49, 1986.

[13] N. Magnenat-Thalmann, et D. Thalmann, *Image Synthesis: Theory and Practice*, Springer-Verlag, Tokyo, 1987.

[14] D.L. Toth, *On Ray Tracing Parametric Surfaces*, Computer Graphics (Proc. SIGGRAPH 85), Vol 19, No. 3, pages 171-179, 1985.

[15] J. V. Uspensky, *Theory of Equations*, McGraw-Hill, New York, 1948.