```
(defsystem glos
  :source-pathname "."
  :source-extension "lisp"
  :binary-pathname nil
  :binary-extension nil

  :components ((:module glos
                        :source-pathname ""
                        :binary-pathname "bin"
                        :components (
                                     (:file "VertexBuffer"
                                            :depends-on ("Render" "Transforms" "
Math3D" "Tools" "Lighting"))
                                     (:file "Render"
                                            :depends-on ("Glos" "Tools"))
                                     (:file "Transforms"
                                            :depends-on ("Math3D"))
                                     (:file "Math3D"
                                            :depends-on ("Tools"))
                                     (:file "Lighting"
                                            :depends-on ("Math3D" "Tools" "Glos"
))
                                     (:file "Tools")
                                     (:file "Glos"))))
  :depends-on nil)
```

```
(in-package :cl-user)

;;; given a system definition, create a single file by concatenating all
;;; the object files that comprise that system - in the correct order

(defun copy-stream (from to)
  "Copy into TO from FROM until end of file, without translating or otherwise mauling anything"
  (let ((buf (make-array 4096 :element-type (stream-element-type from)
                              :initial-element #\Space)))
    (do ((pos (read-sequence buf from)  (read-sequence buf from)))
        ((= 0 pos) nil)
      (write-sequence buf to :end pos))))

(defun print-binary-file-operation (component force)
  "Spit the binary file associated with COMPONENT to *STANDARD-OUTPUT*"
  (with-open-file (i (compile-file-pathname (make::component-pathname component
:binary))
                     :direction :input)
    (copy-stream i *standard-output*))
  nil)

(defun agglomerate-system (system output-file)
  "List files in SYSTEM, in the order that they need to be processed"
  (make::component-operation 'print-binary 'print-binary-file-operation)
  (with-open-file (o output-file :direction :output
                     :if-exists :rename)
    (let ((*standard-output* o))
      (make::operate-on-system  system 'print-binary))))


(defmacro compile-glos (&rest args)
  `(progn (compile-system 'glos ,@args)
          (agglomerate-system 'glos "glos-library.x86f")))


(defmacro compile-ascparser (&rest args)
  `(progn (compile-system 'ascparser ,@args)
          (agglomerate-system 'ascparser "ascparser-library.x86f")))
```

```
(setq sym (intern "DEFINE-KEYSYM"))
(shadowing-import sym 'xlib)
(setq sym (intern "ANGLE"))
(shadowing-import sym 'xlib)
(if (not (member :MK-DEFSYSTEM *FEATURES*))
    (load "library:subsystems/defsystem-library.x86f"))
(if (not (member :XLIB *FEATURES*))
    (load "library:subsystems/clx-library.x86f"))
(use-package :xlib)
(load "./agglomerate-system.lisp")
(declaim (optimize (extensions:inhibit-warnings 3)))
(compile-glos)
(compile-ascparser)
(compile-file "./DemoASC.lisp")
(quit)
```

```
(setq sym (intern "DEFINE−KEYSYM"))
(shadowing−import sym 'xlib)
(setq sym (intern "ANGLE"))
(shadowing−import sym 'xlib)
(if (not (member :MK−DEFSYSTEM *FEATURES*))
    (load "library:subsystems/defsystem−library.x86f"))
(if (not (member :XLIB *FEATURES*))
    (load "library:subsystems/clx−library.x86f"))
(use−package :xlib)
(setf mk::*central−registry* (append (list ".") mk::*central−registry*))
(load "./agglomerate−system.lisp")
(compile−glos)
(compile−ascparser)
(compile−file "./DemoASC.lisp")
(quit)
```

```
(setq sym (intern "DEFINE-KEYSYM"))
(shadowing-import sym 'xlib)
(setq sym (intern "ANGLE"))
(shadowing-import sym 'xlib)
(load "library:subsystems/defsystem-library.x86f")
(load "library:subsystems/clx-library.x86f")
(use-package :xlib)
(load "DemoASC.x86f")
(test-asc)
(quit)
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: DemoASC.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:41 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))



(load "./glos-library.x86f")
(load "./ascparser-library.x86f")


(defun test-asc ( &rest args &key (host "localhost")
                        (string "Asc parser")
                        (font "fixed")
                        (file "asc/dolphin.asc")
                        (zoom 1000.0))
  (let ((display nil)
        (abort t))
    (unwind-protect
        (progn
          (setf display (open-display host))
          (multiple-value-prog1
            (let* ((obj-list (load-list-asc file))
                   (dolphin (car obj-list))

                   (screen (display-default-screen display))
                   (black (screen-black-pixel screen))
                   (white (screen-white-pixel screen))
                   (font (open-font display font))
                   (width *screen-width*)
                   (height *screen-height*)
                   (x (truncate (- (screen-width screen) width) 2))
                   (y (truncate (- (screen-height screen) height) 2))
                   (window (create-window :parent (screen-root screen)
                                          :x x :y y :width width :height height
                                          :background black
                                          :border white
                                          :border-width 1
                                          :colormap (screen-default-colormap scr
een)
                                          :bit-gravity :center
                                          :event-mask '(:exposure :button-press)
))
                   (gcontext (create-gcontext :drawable window
                                              :background black
                                              :foreground white
                                              :font font)))
              ;; Set window manager hints
              (set-wm-properties window
                                 :name 'Test-Begin
                                 :icon-name string
                                 :resource-name string
                                 :resource-class 'Test-GLOS
                                 :command (list* 'Test-GLOS host args)
                                 :x x :y y :width width :height height
```



```lisp
                                 :min-width width :min-height height
                                 :input :off :initial-state :normal)
              (init-glos)
              ;(ViewPort  0 0 160 100)
              (ShadeModel :SMOOTH)

              (Enable :LIGHTING)
              (Enable LIGHT0)
              (Enable LIGHT1)
              (Enable LIGHT2)

              (Light LIGHT0 :POSITION 0.5 0.0 0.5 0.0)
              (Light LIGHT0 :SPECULAR 1.0 1.0 1.0 1.0)

              (Light LIGHT1 :POSITION 0.5 -0.5 0.5 0.0)
              (Light LIGHT1 :DIFFUSE 0.0 0.0 0.0 1.0)
              (Light LIGHT1 :AMBIANT 0.0 0.0 0.0 1.0)
              (Light LIGHT1 :SPECULAR 0.117 0.392 0.392 1.0)

              (Light LIGHT2 :POSITION -0.5 -0.5 -0.5 0.0)
              (Light LIGHT2 :DIFFUSE 0.157 0.157 0.253 1.0)
              (Light LIGHT2 :AMBIANT 0.157 0.157 0.253 1.0)
              (Light LIGHT2 :SPECULAR 0.0 0.0 1.0 1.0)

              (Material :SPECULAR 1.0 1.0 1.0 1.0)
              (Material :SHININESS 128.0)

              (Enable :DEPTH-BUFFER)
              ;(Disable  :NORMALIZE)
              (ClearColor 0 0 0)
              (MatrixMode :PROJECTION)
              (LoadIdentity)
              ;(Frustum -1.0 1.0 -1.0 1.0 1.5 20.0)
              (Perspective 60.0 (coerce (/ *viewport-width* *viewport-height*) '
single-float) 100.0 2000.0)
              (MatrixMode :MODELVIEW)
              (map-window window)
              ;; Handle events
              (event-case (display :discard-p t :force-output-p t)
                          (exposure  ;; Come here on exposure events
                           (window count)
                           (when (zerop count)
                             (dotimes (i 360)
                               (let ((angle (* i (/ 3.14159 180.0))))
                                 (declare (type single-float angle))

                                 (Clear :COLOR-BUFFER :DEPTH-BUFFER)

                                 (LoadIdentity)
                                 (LookAt 0.0 0.0 zoom
                                         0.0 0.0 0.0
                                         0.0 1.0 0.0)
                                 ;(Translate 0.0 0.0 -1500.0)

                                 (Rotate angle 0.0 1.0 0.0)
                                 (Rotate 1.5707963267948966 1.0 0.0 0.0)
                                 (Color 157 157 253)
                                 (Begin-object dolphin)

                                 (put-image window gcontext *image*
                                            :x 0 :y 0 :width *screen-width* :hei
ght *screen-height*)))))
```

```
                              ;; Ignore all but the last exposure event
                              ;; Returning non-nil causes event-case to exit
                           (button-press () t))
               (setf abort nil))))
       ;; Ensure display is closed when done
       (when display
         (close-display display :abort abort)))))
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Glos.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:38 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

;;; Global variables that must be accessible everywhere

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))

(use-package :xlib)

(defparameter *screen-width* 320)
(defparameter *screen-height* 200)

(defparameter *screen-centerx* (truncate *screen-width* 2))
(defparameter *screen-centery* (truncate *screen-height* 2))

(defparameter *near* 10)
(defparameter *far* 2000)


;;; color and depth buffers
(defparameter *imagebuf* (make-array (list *screen-height* *screen-width*) :eleme
nt-type 'card16))
(defparameter *zbuffer* (make-array (list *screen-height* *screen-width*) :element
-type 'int32))
(defparameter *image* (create-image :data *imagebuf* :depth 16))

(declaim (type (simple-array card16) *imagebuf*)
         (type (simple-array int32) *zbuffer*)
         (type card32 *near* *far*)
         (type card16 *screen-width* *screen-height* *screen-centerx* *screen-ce
ntery*))


;;; lighting variables

(defparameter *nblight* 8)
(declaim (type card8 *nblight*))


;;; state variables

(defparameter *clear-color* 0)

(defparameter *do-lighting* nil)
(defparameter *use-zbuffer* t)
(defparameter *do-normals-normalize* nil)

(defparameter *shade-model* :FLAT)

(declaim (type card16 *clear-color*))
(declaim (type symbol *shade-model*))

(defconstant +state-keys+ '(:LIGHTING :DEPTH-BUFFER :NORMALIZE))
(defconstant +state-vars+ '(*do-lighting* *use-zbuffer* *do-normals-normalize*))
```



```lisp
;;; state handling functions

(defmacro make-toggle (macroname bool)
  `(defmacro ,macroname (mode)
     (if (and (numberp (eval mode))
              (>= (eval mode) 0)
              (< (eval mode) *nblight*))
         `(setf (aref *tlightsbool* ,mode) ,,bool)
       (if (not (member mode +state-keys+))
           (error "~S: unknown mode: ~S" (string ',macroname) mode)
         `(setf ,(elt +state-vars+ (position mode +state-keys+)) ,,bool)))))

(make-toggle Enable t)
(make-toggle Disable nil)


(defmacro ShadeModel (model)
  "Set the shade model"
  (if (or (eq model :FLAT) (eq model :SMOOTH))
      `(setf *shade-model* ,model)
    (error "ShadeModel: unknown argument")))



;;; viewport variables

(defvar *viewport-width*)
(defvar *viewport-height*)
(defvar *viewport-half-width*)
(defvar *viewport-half-height*)
(defvar *viewport-centerx*)
(defvar *viewport-centery*)
(declaim (type card16 *viewport-width* *viewport-height* *viewport-half-width* *
viewport-half-height*
               *viewport-centerx* *viewport-centery*))

(defun ViewPort (x y w h)
  "Set the viewport to be a rectangle of width w and height h (in pixels). its
 upper-left corner is at (x,y) in pixels"
  (declare (type card16 x y w h))
  (setf *viewport-width*   w
        *viewport-height*  h
        *viewport-half-width*  (truncate w 2)
        *viewport-half-height* (truncate h 2))
  (setf *viewport-centerx* (+ x *viewport-half-width*)
        *viewport-centery* (+ y *viewport-half-height*)))

(defun init-glos ()
  "Initialize the glos library to default state"
  (Disable :LIGHTING)
  (Disable :NORMALIZE)
  (Disable :DEPTH-BUFFER)
  (ShadeModel :FLAT)
  (ViewPort 0 0 *screen-width* *screen-height*))
```

```
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Tools.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:39 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


;;;---------------------------------------------------------------------------
;;; Useful Macros
;;;---------------------------------------------------------------------------


;;Paul Graham's 'Ansi Common Lisp'
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))


(defmacro progt (&body exps)
  "execute given expressions in sequence and returns T at the end."
  `(progn ,@exps
          t))


;;arguments declaration & update observe the "do" syntax
;;Example: (generate-list 5 (lambda (x y) (+ x y))
;;                          (a 0 (+ a 2)) (b 1 (- b 1)))
;; --> (1 2 3 4 5)
(defmacro generate-list (len genfunc &body paramlist)
  "generate a list of "len" elements, result of the applying the given
lambda to a set of possibly updated arguments"
  (let ((params (mapcar #'car paramlist))
        (k (gensym))
        (acc (gensym)))
    `(do* (,@paramlist
           (,k 0 (1+ ,k))
           (,acc '()))
         ((>= ,k ,len) (reverse ,acc))
       (push (funcall #',genfunc ,@params) ,acc))))


;;Example: (setf myfun (lambda (x y z)
;;            (format t "~%x:~A y:~A z:~A" x y z)))
;;(funcall (lcurry myfun 1 nil 3) 2)
;; --> x:1 y:2 z:3
(defmacro lcurry (lambdaform &body args)
  "returns a curried version of given lambda"
  (let ((myargs '())
        (symlist '()))
    (progn (dolist (x args)
             (if x
                 (push x myargs)
               (let ((sym (gensym)))
                 (push sym symlist)
                 (push sym myargs))))
```



```
           `(lambda ,(nreverse symlist)
              (funcall ,lambdaform ,@(nreverse myargs))))))


;;consider multiple-value-setq for multiple-value assignment
(defmacro nsetf (value &body vars)
  "declare n variables initialized to an unique value."
  (let ((acc '()))
    (dolist (x vars)
      (push x acc)
      (push value acc))
    `(setf ,@(nreverse acc))))


(defmacro list-match-setf (lsrc ldest)
  "setf n variables to n values, all included in separate lists"
  (let ((gencode (lambda (x y) `(setf ,x ,y))))
    `(progn ,@(mapcar gencode lsrc ldest)
            t)))


(defmacro list-match-incf (lsrc ldest)
  "incf n variables of n increments, all included in separate lists"
  (let ((gencode (lambda (x y) `(incf ,x ,y))))
    `(progn ,@(mapcar gencode lsrc ldest)
            t)))


;;;---------------------------------------------------------------------------
;;; Useful functions
;;;---------------------------------------------------------------------------

(defun list-values (l)
  "return values of a list"
  (apply #'values l))

;;Paul Graham's 'Ansi Common Lisp'
(defun curry (fn &rest args)
  #'(lambda (&rest args2)
      (apply fn (append args args2))))

;;Paul Graham's 'On Lisp'
(defun mkstr (&rest args)
  (with-output-to-string (s)
                         (dolist (a args) (princ a s))))

;;Paul Graham's 'On Lisp'
(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))


;;;---------------------------------------------------------------------------
;;; Reader Macros
;;;---------------------------------------------------------------------------


;; reader macro for accessing mat44 elements
;; #[m 2 3] -> 1.0
(progn
  (set-macro-character
   #\] (get-macro-character #\)))
```

```
  (set-dispatch-macro-character
   #\# #\[
   #'(lambda (stream char1 char2)
       (declare (ignore char1 char2))
       (let* ((trike (read-delimited-list #\] stream t))
              (m (car trike))
              (i (cadr trike))
              (j (caddr trike)))
         (if (numberp i)
             (if (numberp j)
                 (let ((index (the fixnum (+ (coerce j 'fixnum) (* (coerce i 'fi
xnum) 4)))))
                   `(the single-float (aref ,m ,index)))
               ; j n'est pas un nobre
               (let ((index (the fixnum (* (coerce i 'fixnum) 4))))
                 `(the single-float (aref ,m (+ ,j ,index)))))
           `(the single-float (aref ,m (+ ,j (the fixnum (* ,i 4))))))))))



;; reader macro for accessing vector slots
(progn
  (set-macro-character
   #\} (get-macro-character #\)))

  (set-dispatch-macro-character
   #\# #\{
   #'(lambda (stream char1 char2)
       (declare (ignore char1 char2))
       (let* ((trike (read-delimited-list #\} stream t))
              (v (car trike))
              (i (cadr trike)))
         `(the single-float (aref ,v ,i))))))


;;;-----------------------------------------------------------------------------
;;; Useful utilities
;;;-----------------------------------------------------------------------------


;; multiple subst in a list:
;; substn '(1 a 2 b 3 c) '(a b 4 c coucou) -> '(1 2 4 3 coucou)
(defun substn (lst body)
  "multiple substitutions in a list"
  (if (null lst)
      body
    (substn (cddr lst)
            (subst (first lst)
                   (second lst)
                   body))))
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Lighting.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:41 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


(defstruct material
  (ambiant   (vec4-make 0.2 0.2 0.2 1.0) :type vec4)
  (diffuse   (vec4-make 0.8 0.8 0.8 1.0) :type vec4)
  (specular  (vec4-make 0.0 0.0 0.0 1.0) :type vec4)
  (shininess 0.0                         :type single-float)
  (emission  (vec4-make 0.0 0.0 0.0 1.0) :type vec4))


(defstruct light
  (ambiant        (vec4-make 0.0 0.0 0.0 1.0) :type vec4)
  (diffuse        (vec4-make 1.0 1.0 1.0 1.0) :type vec4)
  (specular       (vec4-make 1.0 1.0 1.0 1.0) :type vec4)
  (position       (vec4-make 0.0 0.0 1.0 0.0) :type vec4)
  (spot_direction (vec3-make 0.0 0.0 -1.0)    :type vec3)
  (spot_exponent  0.0        :type single-float)
  (sport_cutoff   180.0      :type single-float)
  (constant_attenuation  1.0 :type single-float)
  (linear_attenuation    0.0 :type single-float)
  (quadratic_attenuation 0.0 :type single-float))


(defparameter *current-material* (make-material))
(defparameter *tlights* (make-array *nblight* :element-type 'light))
(defparameter *tlightsbool* (make-array *nblight* :element-type 'boolean))
(declaim (type material *current-material*)
         (type (simple-vector *) *tlights*)
         (type (simple-vector *) *tlightsbool*))


;; allocates memory for the tab light to avoid consing
(dotimes (i *nblight*)
  (declare (type card8 i))
  (setf (aref *tlights* i) (make-light)
        (aref *tlightsbool* i) nil))


(defconstant LIGHT0 0)
(defconstant LIGHT1 1)
(defconstant LIGHT2 2)
(defconstant LIGHT3 3)
(defconstant LIGHT4 4)
(defconstant LIGHT5 5)
(defconstant LIGHT6 6)
(defconstant LIGHT7 7)


(defmacro Light (light-num slot-type &rest slot-values)
  (let ((slot (symb "LIGHT-" slot-type))
        (first-val (first slot-values))
```



```lisp
        (second-val (second slot-values))
        (third-val (third slot-values))
        (fourth-val (fourth slot-values)))
    (case slot-type
      ((:AMBIANT :DIFFUSE :SPECULAR :POSITION)
       `(setf (vec-x (,slot (aref *tlights* ,light-num))) ,first-val
              (vec-y (,slot (aref *tlights* ,light-num))) ,second-val
              (vec-z (,slot (aref *tlights* ,light-num))) ,third-val
              (vec-w (,slot (aref *tlights* ,light-num))) ,fourth-val))
      (:SPOT_DIRECTION
       `(setf (vec-x (,slot (aref *tlights* ,light-num))) ,first-val
              (vec-y (,slot (aref *tlights* ,light-num))) ,second-val
              (vec-z (,slot (aref *tlights* ,light-num))) ,third-val))
      (t
       `(setf (,slot (aref *tlights* ,light-num)) ,first-val)))))


(defmacro Material (slot-type &rest slot-values)
  (let ((slot (symb "MATERIAL-" slot-type))
        (first-val (first slot-values))
        (second-val (second slot-values))
        (third-val (third slot-values))
        (fourth-val (fourth slot-values)))
    (case slot-type
      (:SHININESS
       `(setf (,slot *current-material*) ,first-val))
      (t
       `(setf (vec-x (,slot *current-material*)) ,first-val
              (vec-y (,slot *current-material*)) ,second-val
              (vec-z (,slot *current-material*)) ,third-val
              (vec-w (,slot *current-material*)) ,fourth-val)))))
```

```lisp
;;; ----------------------------------------------------------------------------
;;; $RCSfile: Math3D.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:39 $
;;; $Revision: 1.1.1.1 $
;;; ----------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


(defconstant +PI+ 3.141592653)
(declaim (type single-float +PI+))


(defconstant +mat44-index-list+ '((0 0) (0 1) (0 2) (0 3)
                                  (1 0) (1 1) (1 2) (1 3)
                                  (2 0) (2 1) (2 2) (2 3)
                                  (3 0) (3 1) (3 2) (3 3)))


;;;-----------------------------------------------------------------------------
;;;-----------------------------------------------------------------------------
;; 4x4 Matrices
;;;-----------------------------------------------------------------------------
;;;-----------------------------------------------------------------------------


(deftype mat44 () '(simple-array single-float (16)))


;; Constructor
(defun mat44-make-null ()
  "Create the null matrix"
  (make-array 16 :element-type 'single-float))


;;;----------------------------------------------------
;;; Matrix specific functions
;;;----------------------------------------------------


; i and j must be numbers!
(defmacro mat44-elem (m i j)
  (declare (type card8 i j))
  `(the single-float (aref ,m ,(+ (* 4 i) j))))


(defmacro make-mat44-defuns ()
  "define accessors for the mat44 type"
  (let* ((index +mat44-index-list+)
         (mat-access (lambda (mat ilist) `(#[,mat ,(car ilist) ,(cadr ilist)])))
         (set-zero (lambda (x) `(setf ,@x 0.0)))
         (accessors (mapcar (curry mat-access 'm) index))
         (mtarget (mapcar (curry mat-access 'mtarget) index))
         (msource (mapcar (curry mat-access 'msource) index))
         (mat-slots (mapcar (lambda (ilist) (symb 'M (car ilist) (cadr ilist)))
                            index)))
    `(progt (defun mat44-set-null (m)
              "Modify a matrix which become a null matrix"
```



```lisp
              (declare (type mat44 m))
              (progt ,@(mapcar set-zero accessors)))

            (defun mat44-copy (mtarget msource)
              "copy the matrix msource into mtarget"
              (declare (type mat44 m))
              (progt ,@(mapcar (lambda (x y) `(setf ,@x ,@y))
                               mtarget msource)))

            (defun mat44-set (m ,@mat-slots)
              "fill a matrix with elements"
              (progt ,@(mapcar (lambda (x y) `(setf ,@x ,y))
                               accessors mat-slots))))))


;;effective definition of the accessors in the project
(make-mat44-defuns)


(defun mat44-make-dup (m)
  "duplicate a matrix"
  (declare (type mat44 m))
  (copy-seq m))


(defun mat44-row (matrix rownum)
  "return a list, copy of the nth row of the given matrix"
  (generate-list 4
                 (lambda (x) #[matrix rownum x])
                 (j 0 (1+ j))))


(defun mat44-column (matrix colnum)
  "return a list, copy of the nth column of the given matrix"
  (generate-list 4
                 (lambda (x) #[matrix x colnum])
                 (i 0 (1+ i))))


(defun mat44-index-list ()
  "generate the list of indexes for the mat44 matrix type"
  (do ((i 0 (1+ i))
       (acc '()))
      ((>= i 4) (nreverse acc))
    (declare (type card8 i))
    (dotimes (j 4)
      (push `(,i ,j) acc))))


;;pretty-printing facility for displaying mat44 objects
(set-pprint-dispatch
 'mat44
 #'(lambda (s mat)
     (format s "~%~W~%~W~%~W~%~W
GLOS-MATRIX[4*4] OF SINGLE-FLOAT~%"
             (mat44-row mat 0)
             (mat44-row mat 1)
             (mat44-row mat 2)
             (mat44-row mat 3))))


;;;-----------------------------------------------------------------------------
```

```lisp
;;;-----------------------------------------------------------------------------
;; Vectors
;;;-----------------------------------------------------------------------------
;;;-----------------------------------------------------------------------------


;;;-----------------------------------------------------------------------------
;; Constructors and Accessors
;;;-----------------------------------------------------------------------------

;We have both 3 and 4 elements vector types
;Functions or macros names prefixed with "vec-" take normally vec3 arguments. Bu
t for commodity,
;if a vec4 is passed the fourth coordinate w is ignored and the vec4 is treated
as a vec3.

(deftype vec3 () '(simple-array single-float (3)))
(deftype vec4 () '(simple-array single-float (4)))


;; Constructors
(defun vec3-make-null ()
  (make-array 3 :element-type 'single-float))


(defun vec3-make (x y z)
  (declare (type single-float x y z))
  (make-array 3 :element-type 'single-float :initial-contents (list x y z)))


(defun vec4-make-null ()
  (make-array 4 :element-type 'single-float))


(defun vec4-make (x y z w)
  (declare (type single-float x y z w))
  (make-array 4 :element-type 'single-float :initial-contents (list x y z w)))


;;;-----------------------------------------------------------------------------
;; Vector arithmetic operations
;;;-----------------------------------------------------------------------------

(declaim (inline vec3-set vec3-cpy
                 vec4-set vec4-cpy))

(defun vec3-set (v x y z)
  "Set coordinates of a vec3"
  (declare (type vec3 v)
           (type single-float x y z))
  (setf #{v 0} x
        #{v 1} y
        #{v 2} z)
  t)


(defun vec3-cpy (v1 v2)
  "copy v2 (src) to v1 (dst)"
  (declare (type vec3 v1 v2))
  (setf #{v1 0} #{v2 0}
        #{v1 1} #{v2 1}
        #{v1 2} #{v2 2})
```



```lisp
  t)


(defun vec4-set (v x y z w)
  "Set coordinates of a vec4"
  (declare (type vec4 v)
           (type single-float x y z w))
  (setf #{v 0} x
        #{v 1} y
        #{v 2} z
        #{v 3} w)
  t)


(defun vec4-cpy (v1 v2)
  "copy v2 (src) to v1 (dst)"
  (declare (type vec4 v1 v2))
  (setf #{v1 0} #{v2 0}
        #{v1 1} #{v2 1}
        #{v1 2} #{v2 2}
        #{v1 3} #{v2 3})
  t)


;;these accessors are macros because we want them 'setable'
(defmacro vec-x (v)
  `(aref ,v 0))
(defmacro vec-y (v)
  `(aref ,v 1))
(defmacro vec-z (v)
  `(aref ,v 2))
(defmacro vec-w (v)
  `(aref ,v 3))


(defmacro vec-norm (v)
  "Return the length of the argument vector"
  `(the single-float (sqrt (the single-float (+ (* #{,v 0} #{,v 0})
                                                (* #{,v 1} #{,v 1})
                                                (* #{,v 2} #{,v 2}))))))


(defmacro vec-dot-product (v1 v2)
  "Return the dot product of the argument vectors"
  `(the single-float (+ (* #{,v1 0} #{,v2 0})
                        (* #{,v1 1} #{,v2 1})
                        (* #{,v1 2} #{,v2 2}))))


;;pretty-printing facility for displaying vec objects
(set-pprint-dispatch
 'vec3
 #'(lambda (s vec)
     (format s "~%{ ~W; ~W; ~W }
GLOS-VECTOR-3 OF SINGLE-FLOAT~%"
             (vec-x vec)
             (vec-y vec)
             (vec-z vec))))


(set-pprint-dispatch
 'vec4
```

```
 #'(lambda (s vec)
     (format s "~%{ ~W; ~W; ~W; ~W }
GLOS-VECTOR-4 OF SINGLE-FLOAT~%"
             (vec-x vec)
             (vec-y vec)
             (vec-z vec)
             (vec-w vec))))


(defmacro vec-set-cross-product (vtarget v1 v2)
  "vtarget become the result of the cross product between v1 and v2"
  `(vec3-set ,vtarget
             (- (* #{,v1 1} #{,v2 2}) (* #{,v1 2} #{,v2 1}))
             (- (* #{,v1 2} #{,v2 0}) (* #{,v1 0} #{,v2 2}))
             (- (* #{,v1 0} #{,v2 1}) (* #{,v1 1} #{,v2 0}))))


(defmacro vec-set-normalize (v)
  "normalized the vector v and return v"
  (let ((n (gensym)))
    `(let ((,n (vec-norm ,v)))
       (declare (type single-float ,n))
       (if (not (= ,n 0.0))
           (vec3-set ,v (/ #{,v 0} ,n) (/ #{,v 1} ,n) (/ #{,v 2} ,n))))))
```

```
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Transforms.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:40 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


;;;---------------------------------------------------------------------------
;;; Declaration
;;;---------------------------------------------------------------------------


;matrix and vector pools for transform primitives
(defparameter +transforms-matrix-pool+ (make-array 16 :fill-pointer t))
(defparameter +transforms-vector-pool+ (make-array 16 :fill-pointer t))
(defparameter +transforms-matrix-stack-pool+ (make-array 64 :fill-pointer t))


(defparameter *projection-matrix* (mat44-make-null))
(defparameter *modelview-matrix*  (mat44-make-null))
(defparameter *current-matrix* *modelview-matrix*)
(defparameter *current-mode* :MODELVIEW)

;matrix stacks for PushMatrix/PopMatrix
(defparameter *projection-matrix-stack* (make-array 32 :fill-pointer 0))
(defparameter *modelview-matrix-stack* (make-array 32 :fill-pointer 0))


(declaim (type vector
               +transforms-matrix-pool+
               +transforms-vector-pool+
               +transforms-matrix-stack-pool+
               *projection-matrix-stack*
               *modelview-matrix-stack*)
         (type mat44
               *current-matrix*
               *projection-matrix*
               *modelview-matrix*))


;fill pools arrays
(dotimes (i 16)
  (setf (aref +transforms-matrix-pool+ i) (mat44-make-null)
        (aref +transforms-vector-pool+ i) (vec4-make-null)))
(dotimes (i 64)
  (setf (aref +transforms-matrix-stack-pool+ i) (mat44-make-null)))


;;;---------------------------------------------------------------------------
;;; Matrix stacks stuff
;;;---------------------------------------------------------------------------


(defun stack-of-current-matrix ()
  "Return the matrix stack of the current mode"
  (if (eq *current-matrix* *modelview-matrix*)
```



```
      *modelview-matrix-stack*
    *projection-matrix-stack*))


(defun matrix-of-current-matrix ()
  (if (eq *current-matrix* *modelview-matrix*)
      '*modelview-matrix*
    '*projection-matrix*))


(defmacro MatrixMode (mode)
  "Sets the current matrix mode for future operations on matrices"
  (case mode
    (:MODELVIEW  `(setf *current-matrix* *modelview-matrix*))
    (:PROJECTION `(setf *current-matrix* *projection-matrix*))
    (t (error "Unknown matrix mode: ~S" mode))))


(defun PushMatrix ()
  "Push the current-matrix on top of the matrix stack"
  (let ((temp-matrix *current-matrix*)
        (matrix-of-current-mode (matrix-of-current-matrix)))
    (declare (type mat44 temp-matrix)
             (type symbol matrix-of-current-mode))
    (vector-push *current-matrix* (stack-of-current-matrix))
    (setf *current-matrix* (vector-pop +transforms-matrix-stack-pool+))
    (mat44-copy *current-matrix* temp-matrix)
    (set matrix-of-current-mode *current-matrix*)
    t))


(defun PopMatrix ()
  "Set the current matrix to the matrix on top of the matrix stack"
  (let ((matrix-of-current-mode (matrix-of-current-matrix)))
    (declare (type symbol matrix-of-current-mode))
    (vector-push *current-matrix* +transforms-matrix-stack-pool+)
    (setf *current-matrix* (vector-pop (stack-of-current-matrix)))
    (set matrix-of-current-mode *current-matrix*)
    t))

;;;---------------------------------------------------------------------------
;;; Matrix Transformations
;;;---------------------------------------------------------------------------


; on ne peut pas utiliser le reader macro ici a cause de l'ordre de la macroexpa
nsion:
; la reader macro est toujours expansee *avant* le reste de la macro. donc on ut
ilise mat44-elem.
(defmacro row-by-col (mat-dest mat1 mat2 i j)
  (declare (type card8 i j))
  `(setf (mat44-elem ,mat-dest ,i ,j)
         (the single-float (+ (the single-float (* (mat44-elem ,mat1 ,i 0) (mat4
4-elem ,mat2 0 ,j)))
                              (the single-float (* (mat44-elem ,mat1 ,i 1) (mat4
4-elem ,mat2 1 ,j)))
                              (the single-float (* (mat44-elem ,mat1 ,i 2) (mat4
4-elem ,mat2 2 ,j)))
                              (the single-float (* (mat44-elem ,mat1 ,i 3) (mat4
4-elem ,mat2 3 ,j)))))))
```

```
(defmacro calc-line (mat-dest mat1 mat2 i)
  (declare (type card8 i))
  `(progn (row-by-col ,mat-dest ,mat1 ,mat2 ,i 0)
          (row-by-col ,mat-dest ,mat1 ,mat2 ,i 1)
          (row-by-col ,mat-dest ,mat1 ,mat2 ,i 2)
          (row-by-col ,mat-dest ,mat1 ,mat2 ,i 3)))


(defun MultMatrix (mat)
  "Multiply the current matrix by the argument matrix and store the result into the current matrix"
   (let ((m (vector-pop +transforms-matrix-pool+)))
     (declare (type mat44 mat m))
     (mat44-copy m *current-matrix*)
     (calc-line *current-matrix* m mat 0)
     (calc-line *current-matrix* m mat 1)
     (calc-line *current-matrix* m mat 2)
     (calc-line *current-matrix* m mat 3)
     (vector-push m +transforms-matrix-pool+)))


(defmacro LoadMatrix (m)
  "Set the current matrix to an arbitrary matrix"
  `(mat44-copy *current-matrix* ,m))


(defun LoadIdentity ()
  "Set the current matrix to the identity matrix"
  (mat44-set *current-matrix*
             1.0 0.0 0.0 0.0
             0.0 1.0 0.0 0.0
             0.0 0.0 1.0 0.0
             0.0 0.0 0.0 1.0)
  t)


(defun Translate (x y z)
  (declare (type single-float x y z))
  "Multiply the current matrix by a translation matrix"
  (setf #[*current-matrix* 0 3] (+ (* x #[*current-matrix* 0 0])
                                   (* y #[*current-matrix* 0 1])
                                   (* z #[*current-matrix* 0 2])
                                   #[*current-matrix* 0 3])
        #[*current-matrix* 1 3] (+ (* x #[*current-matrix* 1 0])
                                   (* y #[*current-matrix* 1 1])
                                   (* z #[*current-matrix* 1 2])
                                   #[*current-matrix* 1 3])
        #[*current-matrix* 2 3] (+ (* x #[*current-matrix* 2 0])
                                   (* y #[*current-matrix* 2 1])
                                   (* z #[*current-matrix* 2 2])
                                   #[*current-matrix* 2 3])
        #[*current-matrix* 3 3] (+ (* x #[*current-matrix* 3 0])
                                   (* y #[*current-matrix* 3 1])
                                   (* z #[*current-matrix* 3 2])
                                   #[*current-matrix* 3 3]))
  t)


(defun Scale (x y z)
  "Multiply the current matrix by a scaling matrix"
  (declare (type single-float x y z))
  (setf #[*current-matrix* 0 0] (* x #[*current-matrix* 0 0])
        #[*current-matrix* 0 1] (* y #[*current-matrix* 0 1])
```



```
        #[*current-matrix* 0 2] (* z #[*current-matrix* 0 2])
        #[*current-matrix* 1 0] (* x #[*current-matrix* 1 0])
        #[*current-matrix* 1 1] (* y #[*current-matrix* 1 1])
        #[*current-matrix* 1 2] (* z #[*current-matrix* 1 2])
        #[*current-matrix* 2 0] (* x #[*current-matrix* 2 0])
        #[*current-matrix* 2 1] (* y #[*current-matrix* 2 1])
        #[*current-matrix* 2 2] (* z #[*current-matrix* 2 2])
        #[*current-matrix* 3 0] (* x #[*current-matrix* 3 0])
        #[*current-matrix* 3 1] (* y #[*current-matrix* 3 1])
        #[*current-matrix* 3 2] (* z #[*current-matrix* 3 2]))
  t)


(defun Rotate (angle x y z)
  "Multiply the current matrix by a rotation matrix (rotation of angle around the vector (x,y,z)"
  (declare (type single-float angle x y z))
  (let* ((m (vector-pop +transforms-matrix-pool+))
          (cosine (cos angle))
          (sine (sin angle))
          (cos-diff (- 1 cosine))
          (xy (* x y))
          (xz (* x z))
          (yz (* y z))
          (xs (* x sine))
          (ys (* y sine))
          (zs (* z sine)))
    (declare (type single-float cosine sine cos-diff xy xz yz xs ys zs)
             (type mat44 m))
    (mat44-set m
               (+ (* x x cos-diff) cosine)
               (- (* xy cos-diff) zs)
               (+ (* xz cos-diff) ys)
               0.0
               (+ (* xy cos-diff) zs)
               (+ (* y y cos-diff) cosine)
               (- (* yz cos-diff) xs)
               0.0
               (- (* xz cos-diff) ys)
               (+ (* yz cos-diff) xs)
               (+ (* z z cos-diff) cosine)
               0.0
               0.0 0.0 0.0 1.0)
    (MultMatrix m)
    (vector-push m +transforms-matrix-pool+)))


(defun Frustum (left right bottom top near far)
  "Set the frustum"
  (declare (type single-float left right bottom top near far))
  (let*  ((mrl (- right left))
         (mtb (- top bottom))
         (mfn (- far near))
         (m00 (/ (* 2.0 near) mrl))
         (m02 (/ (+ right left) mrl))
         (m11 (/ (* 2.0 near) mtb))
         (m12 (/ (+ top bottom) mtb))
         (m22 (/ (- (+ far near)) mfn))
         (m23 (/ (- (* 2.0 far near)) mfn))
         (frustum-matrix (vector-pop +transforms-matrix-pool+)))
    (declare (type single-float mrl mtb mfn m00 m02 m11 m12 m22 m23)
             (type mat44 frustum-matrix))
    (setf *near* (the card32 (truncate (the single-float near))))
```

```
    (setf *far* (the card32 (truncate (the single-float far))))
    (mat44-set frustum-matrix
               m00 0.0 m02 0.0
               0.0 m11 m12 0.0
               0.0 0.0 m22 m23
               0.0 0.0 -1.0 0.0)
    (MultMatrix frustum-matrix)
    (vector-push frustum-matrix +transforms-matrix-pool+)))


(defun Perspective (fovy aspect znear zfar)
  "Another way to set the frustum using the fov"
  (declare (type single-float fovy aspect znear zfar))
  (let* ((ymax (* znear (tan (* fovy (/ +PI+ 360.0)))))
         (ymin (- ymax)))
    (declare (type single-float ymin ymax))
    (Frustum (* ymin aspect) (* ymax aspect) ymin ymax znear zfar)))


(defun LookAt (eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ)
  "Set the viewer to (eyeX,eyeY,eyeZ) looking at (centerX,centerY,centerZ) with up vector (upX,upY,upZ)"
  (declare (type single-float eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ
))
  (let ((X (vector-pop +transforms-vector-pool+))
        (Y (vector-pop +transforms-vector-pool+))
        (Z (vector-pop +transforms-vector-pool+))
        (m (vector-pop +transforms-matrix-pool+)))
    (declare (type vec4 X Y Z)
             (type mat44 m))
    (vec4-set Z (- eyeX centerX) (- eyeY centerY) (- eyeZ CenterZ) 1.0)
    (vec-set-normalize Z)
    (vec4-set Y upX upY upZ 1.0)
    (vec-set-cross-product X Y Z)
    (vec-set-cross-product Y Z X)
    (vec-set-normalize X)
    (vec-set-normalize Y)
    (mat44-set m
               #{X 0} #{Y 0} #{Z 0} 0.0
               #{X 1} #{Y 1} #{Z 1} 0.0
               #{X 2} #{Y 2} #{Z 2} 0.0
               0.0    0.0    0.0    1.0)
    (MultMatrix m)
    (Translate (- eyeX) (- eyeY) (- eyeZ))
    (vector-push X +transforms-vector-pool+)
    (vector-push Y +transforms-vector-pool+)
    (vector-push Z +transforms-vector-pool+)
    (vector-push m +transforms-matrix-pool+)))
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Render.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:39 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

;;; Drawing routines that affect the framebuffer (*imagebuf*)

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


(defmacro swap (a b)
  (let ((tmp (gensym)))
    `(prog ((,tmp ,a))
           (setf ,a ,b)
           (setf ,b ,tmp))))


(declaim (inline setpix ClearColor))


(defun rgb-to-pixel-32 (r g b)
  (declare (type card8 r g b))
  (logior (logior (ash  r 16) (ash g 8)) b))


(defun rgb-to-pixel-16 (r g b)
  (declare (type card8 r g b))
  (logior (logior (ash  (ash r -3) 11) (ash (ash g -2) 5)) (ash b -3)))


(defun ClearColor (r g b)
  (setf *clear-color* (rgb-to-pixel-16 r g b)))


(defmacro make-clear-fun (funname &rest body)
  `(defun ,funname ()
     (do ((y 0 (1+ y)))
         ((>= y *screen-height*))
       (declare (type card16 y))
       (do ((x 0 (1+ x)))
           ((>= x *screen-width*))
         (declare (type card16 x))
         ,@body))))


(make-clear-fun clear-color-buffer (setf (aref *imagebuf* y x) *clear-color*))
(make-clear-fun clear-depth-buffer (setf (aref *zbuffer* y x) 0))
(make-clear-fun clear-color-and-depth-buffers
                (setf (aref *imagebuf* y x) *clear-color*)
                (setf (aref *zbuffer* y x) 0))


(defmacro Clear (&rest buffers)
  (if (and (member :COLOR-BUFFER buffers) (member :DEPTH-BUFFER buffers))
      `(clear-color-and-depth-buffers)
    (if (member :COLOR-BUFFER buffers)
        `(clear-color-buffer)
```



```lisp
      (if (member :DEPTH-BUFFER buffers)
          `(clear-depth-buffer)
        (error "Clear: unknown buffer type")))))


(defun setpix (x y r g b)
  (declare (type card16 x y)
           (type card8 r g b))
  (let ((color (rgb-to-pixel-16 r g b)))
    (setf (aref *imagebuf* y x) color)))


(defun setpixz (x y z r g b)
  (declare (type card16 x y)
           (type int32 z)
           (type card8 r g b))
  (let ((color (rgb-to-pixel-16 r g b)))
    (when (>= z 0)
      (setf z (the int32 (- (the int32 z) (the int32 *far*))))
      (when (<= z (aref *zbuffer* y x))
        (setf (aref *imagebuf* y x) color
              (aref *zbuffer* y x) z)))
    nil))


(defun rgbline (x1 y1 x2 y2 r g b)
  "Draw line without clipping using Luca algorithm."
  (declare (type int32 x1 y1 x2 y2)
           (type card8 r g b))
  ;; begin push clipping
  (when (< x1 0) (setf x1 0))
  (when (< x2 0) (setf x2 0))
  (when (< y1 0) (setf y1 0))
  (when (< y2 0) (setf y2 0))

  (when (>= x1 *screen-width*) (setf x1 (- *screen-width* 1)))
  (when (>= x2 *screen-width*) (setf x2 (- *screen-width* 1)))
  (when (>= y1 *screen-height*) (setf y1 (- *screen-height* 1)))
  (when (>= y2 *screen-height*) (setf y2 (- *screen-height* 1)))
  ;; end push clipping

  (let ((x x1)
        (y y1)
        (dx (- x2 x1))
        (dy (- y2 y1))
        (adx 0)
        (ady 0)
        (incx 0)
        (incy 0)
        (cumul 0))
    (declare (type int32 x y)
             (type int32 dx dy adx ady incx incy cumul))

    ;; print the last point
    (setpix x2 y2 r g b)

    ;; compute increment
    (if (< dx 0)
        (setf adx (- dx) incx -1)
      (setf adx dx incx 1))

    (if (< dy 0)
```

```
        (setf ady (- dy) incy -1)
      (setf ady dy incy 1))

    ;; start draw line
    (if (> adx ady)
        (progn
          (setf cumul (ash adx -1))
          (do ((i 0 (1+ i)))
              ((>= i adx))
            (declare (type card16 i))
            (setpix x y r g b)
            (incf x incx)
            (incf cumul ady)
            (when (>= cumul adx)
              (decf cumul adx)
              (incf y incy))))
      (progn
        (setf cumul (ash ady -1))
        (do ((i 0 (1+ i)))
            ((>= i ady))
          (declare (type card16 i))
          (setpix x y r g b)
          (incf y incy)
          (incf cumul adx)
          (when (>= cumul ady)
            (decf cumul ady)
            (incf x incx)))))))



;;polygon-filling functions
(macrolet
    ;;function-specific macros
    ((fillpoly3-do-action1 () '(line_h (ash xd -8) (ash xf -8) l))
     (fillpoly3-do-action2 () '(line_h (ash xf -8) (ash xd -8) l))

     (fillpoly3gour-do-action1 () '(progn (line_hg (ash xd -8) (ash rd -8) (ash
gd -8) (ash bd -8)
                                                    (ash xf -8) (ash rf -8) (ash
gf -8) (ash bf -8) l)
                                          (list-match-incf (rd  gd  bd  rf  gf 
bf)
                                                            (d1r d1g d1b d2r d2g
d2b))))

     (fillpoly3gour-do-action2 () '(progn (line_hg (ash xf -8) (ash rf -8) (ash
gf -8) (ash bf -8)
                                                    (ash xd -8) (ash rd -8) (ash
gd -8) (ash bd -8) l)
                                          (list-match-incf (rd  gd  bd  rf  gf 
bf)
                                                            (d1r d1g d1b d2r d2g
d2b))))

     (fillpoly3z-do-action1 () '(progn (line_hz (ash xd -8) (ash xf -8)
                                                 (ash zd -8) (ash zf -8) l)
                                        (incf zd deltaz1)
                                        (incf zf deltaz2)))

     (fillpoly3z-do-action2 () '(progn (line_hz (ash xf -8) (ash xd -8)
                                                 (ash zf -8) (ash zd -8) l)
                                        (incf zd deltaz1)
```



```
                                        (incf zf deltaz2)))

     (fillpoly3zgour-do-action1 () '(progn (line_hzg (ash xd -8) (ash zd -8) (as
h rd -8) (ash gd -8) (ash bd -8)
                                                      (ash xf -8) (ash zf -8) (as
h rf -8) (ash gf -8) (ash bf -8) l)
                                           (list-match-incf (zd      zf      rd 
 gd  bd  rf  gf  bf)
                                                             (deltaz1 deltaz2 d1r
 d1g d1b d2r d2g d2b))))

     (fillpoly3zgour-do-action2 () '(progn (line_hzg (ash xf -8) (ash zf -8) (as
h rf -8) (ash gf -8) (ash bf -8)
                                                      (ash xd -8) (ash zd -8) (as
h rd -8) (ash gd -8) (ash bd -8) l)
                                           (list-match-incf (zd      zf      rd 
 gd  bd  rf  gf  bf)
                                                             (deltaz1 deltaz2 d1r
 d1g d1b d2r d2g d2b)))))

  (macrolet
      ;;common macros
      ((abc-swap (&body swapground)
                 (let* ((abc-comb '((a b) (a c) (b c)))
                        (swapcode (lambda (x a b) `(swap ,(symb x a) ,(symb x b)
)))
                        (whencode (lambda (a b) `(when (> ,(symb 'Y a) ,(symb 'Y
 b))
                                                      ,@(mapcar (lcurry swapcode ni
l a b) swapground))))
                        (mapwhen (lambda (l) (funcall whencode (car l) (cadr l))
)))
                   `(progn ,@(mapcar mapwhen abc-comb))))


       (if-delta (dnum1 dnum2 ynum1 ynum2 do-action1 do-action2)
                 (let* ((delta1 (symb 'DELTA dnum1))
                        (delta2 (symb 'DELTA dnum2))
                        (y1 (symb 'Y ynum1))
                        (y2 (symb 'Y ynum2))
                        (gen-do (lambda (do-action)
                                  `(do ((l ,y1 (1+ l)))
                                       ((>= l ,y2))
                                     (declare (type int32 l))
                                     (,do-action)
                                     (incf xd delta1)
                                     (incf xf delta2)))))
                   `(if (< ,delta1 ,delta2)
                        ,(funcall gen-do do-action1)
                      ,(funcall gen-do do-action2)))))

    (flet
        ;;common functions
        ((when-delta1 (x1 x2 y1 y2)
                      (truncate (the int32 (ash (the int32 (- x1 x2)) 8))
                                (the int32 (- y1 y2))))

         (when-d (r1 r2 y1 y2) (truncate (ash (- r1 r2) 8) (the int32 (- y1 y2))
)))
      (declare (inline when-delta1)
               (inline when-d))
```

```lisp
      ;;FIRST DEFINED FUNCTION
      (defun fillpoly3 (xa ya xb yb xc yc r g b)
        "Fill a triangle with uniform color."
        (declare (type int32 xa ya xb yb xc yc)
                 (type card8 r g b))

        (flet ((line_h (xd xf h)
                       (declare (type int32 xd xf h))
                       ;; clipping haut et bas
                       (when (or (< h 0) (>= h *screen-height*)) (return-from li
ne_h))
                       ;; clipping gauche
                       (when (< xd 0) (setf xd 0))
                       ;; clipping droit
                       (when (>= xf *screen-width*) (setf xf (- *screen-width* 1
)))

                       (do ((i xd (1+ i)))
                           ((> i xf))
                         (declare (type card16 i))
                         (setpix i h r g b))))

          (let ((delta1 0)
                (delta2 0)
                (xd 0)
                (xf 0))
            (declare (type int32 delta1 delta2 xd xf))

            ;; trie des y
            (abc-swap X Y)

            ;; remplissage
            (when (/= ya yc) (setf delta1 (the int32 (when-delta1 xa xc ya yc)))
)
            (when (/= ya yb) (setf delta2 (the int32 (when-delta1 xb xa yb ya)))
)

            (setf xd (ash (the int32  xa) 8))
            (setf xf xd)

            (if-delta 1 2 A B fillpoly3-do-action1 fillpoly3-do-action2)

            (when (/= yb yc) (setf delta2 (the int32 (when-delta1 xb xc yb yc)))
                  (setf xf (ash (the int32 xb) 8))
                  (if-delta 2 1 B C fillpoly3-do-action1 fillpoly3-do-action2)))
))


      ;;SECOND DEFINED FUNCTION
      (defun fillpoly3gour (xa ya ra ga ba xb yb rb gb bb xc yc rc gc bc)
        "Fill a triangle with gouraud shading."
        (declare (type int32 xa ya xb yb xc yc)
                 (type card8 ra ga ba rb gb bb rc gc bc))

        (flet ((line_hg (xd rd gd bd xf rf gf bf h)
                        (declare (type int32 xd xf h)
                                 (type int32 rd gd bd rf gf bf))
                        (let ((ir 0)
                              (ig 0)
                              (ib 0)
```



```lisp
                              (cr (ash rd 8))
                              (cg (ash gd 8))
                              (cb (ash bd 8)))
                          (declare (type int32 ir ig ib cr cg cb))
                          ;; clipping haut et bas
                          (when (or (< h 0) (>= h *screen-height*)) (return-from
 line_hg))

                          (if (= xd xf)
                              (setf ir 0
                                    ig 0
                                    ib 0)
                            (setf ir (when-delta1 rf rd xf xd)
                                  ig (when-delta1 gf gd xf xd)
                                  ib (when-delta1 bf bd xf xd)))

                          ;; clipping droit
                          (when (>= xf *screen-width*) (setf xf (- *screen-width
* 1)))
                          (do ((i xd (1+ i)))
                              ((> i xf))
                            (declare (type int32 i))
                            (if (>= i 0) (setpix i h
                                                 (the card8 (ash cr -8))
                                                 (the card8 (ash cg -8))
                                                 (the card8 (ash cb -8))))
                            (incf cr ir)
                            (incf cg ig)
                            (incf cb ib)))))


          (let ((delta1 0)
                (delta2 0)
                (d1r 0) (d1g 0) (d1b 0) ; delta1 color
                (d2r 0) (d2g 0) (d2b 0) ; delta2 color
                (rd 0) (gd 0) (bd 0)    ; debut color
                (rf 0) (gf 0) (bf 0)    ; fin color
                (xd 0)
                (xf 0))

            (declare (type int32 delta1 delta2
                           d1r d1g d1b
                           d2r d2g d2b
                           rd gd bd
                           rf gf bf
                           xd xf))

            ;; trie des y
            (abc-swap X Y R G B)

            ;; remplissage
            (when (/= ya yc)
              (setf delta1 (the int32 (when-delta1 xa xc ya yc))
                    d1r (when-d ra rc ya yc)
                    d1g (when-d ga gc ya yc)
                    d1b (when-d ba bc ya yc)))

            (when (/= ya yb)
              (setf delta2 (the int32 (when-delta1 xb xa yb ya))
                    d2r (when-d rb ra yb ya)
                    d2g (when-d gb ga yb ya)
                    d2b (when-d bb ba yb ya)))
```

```
            (setf xd (ash (the int32  xa) 8))
            (setf rd (ash ra 8))
            (setf gd (ash ga 8))
            (setf bd (ash ba 8))
            (list-match-setf (xf rf gf bf) (xd rd gd bd))

            (if-delta 1 2 A B fillpoly3gour-do-action1 fillpoly3gour-do-action2)

            (when (/= yb yc)
              (setf delta2 (the int32 (when-delta1 xb xc yb yc)))
              (setf xf (ash (the int32 xb) 8))
              (setf d2r (when-d rb rc yb yc))
              (setf d2g (when-d gb gc yb yc))
              (setf d2b (when-d bb bc yb yc))
              (setf rf (ash rb 8))
              (setf gf (ash gb 8))
              (setf bf (ash bb 8))

              (if-delta 2 1 B C fillpoly3gour-do-action1 fillpoly3gour-do-action
2)))))


      ;;THIRD DEFINED FUNCTION
      (defun fillpoly3z (xa ya za xb yb zb xc yc zc r g b)
        "fill a triangle with uniform color and zbuffer"
        (declare (type int32 xa ya za xb yb zb xc yc zc)
                 (type card8 r g b))
        (flet ((line_hz (xd xf zd zf h)
                        (declare (type int32 xd zd xf zf h))
                        (let ((deltaz 0);; delta z
                              (curz 0));; current z
                          (declare (type int32 deltaz curz))
                          ;; clipping haut et bas
                          (when (or (< h 0) (>= h *screen-height*)) (return-from
 line_hz))
                          ;; clipping gauche
                          (when (< xd 0) (setf xd 0))
                          ;; clipping droit
                          (when (>= xf *screen-width*) (setf xf (- *screen-width
* 1)))
                          ;; erreur de sens
                          (when (> xd xf) (return-from line_hz))

                          (when (/= xd xf)
                            (setf deltaz (when-delta1 zf zd xf xd)))
                          (setf curz (ash zd 8))

                          (do ((i xd (1+ i)))
                              ((> i xf))
                            (declare (type card16 i))
                            (setpixz i h (ash curz -8) r g b)
                            (incf curz deltaz)))))

          (let ((delta1 0)
                (delta2 0)
                (deltaz1 0)
                (deltaz2 0)
                (xd 0)
                (xf 0)
                (zd 0)
                (zf 0))
```



```
            (declare (type int32 delta1 delta2 deltaz1 deltaz2 xd xf zd zf))

            ;; trie des y
            (abc-swap x y z)

            ;; remplissage
            (when (/= ya yc)
              (setf delta1 (the int32 (when-delta1 xa xc ya yc))
                    deltaz1 (the int32 (when-delta1 za zc ya yc))))

            (when (/= ya yb)
              (setf delta2 (the int32 (when-delta1 xb xa yb ya))
                    deltaz2 (the int32 (when-delta1 zb za yb ya))))

            (setf xd (ash (the int32 xa) 8))
            (setf xf xd)
            (setf zd (ash (the int32 za) 8))
            (setf zf zd)

            (if-delta 1 2 A B fillpoly3z-do-action1 fillpoly3z-do-action2)

            (when (/= yb yc)
              (setf delta2 (the int32 (when-delta1 xb xc yb yc)))
              (setf deltaz2 (the int32 (when-delta1 zb zc yb yc)))
              (setf xf (ash (the int32 xb) 8))
              (setf zf (ash (the int32 zb) 8))

              (if-delta 2 1 B C fillpoly3z-do-action1 fillpoly3z-do-action2)))))


      ;;FOURTH DEFINED FUNCTION
      (defun fillpoly3zgour (xa ya za ra ga ba
                             xb yb zb rb gb bb
                             xc yc zc rc gc bc)
        "Fill a triangle with gouraud shading."
        (declare (type int32 xa ya za xb yb zb xc yc zc)
                 (type card8 ra ga ba rb gb bb rc gc bc))

        (flet ((line_hzg (xd zd rd gd bd xf zf rf gf bf h)
                         (declare (type int32 xd zd xf zf h)
                                  (type int32 rd gd bd rf gf bf))
                         (let ((deltaz 0)
                               (curz 0)
                               (ir 0)
                               (ig 0)
                               (ib 0)
                               (cr (ash rd 8))
                               (cg (ash gd 8))
                               (cb (ash bd 8)))
                           (declare (type int32 deltaz curz ir ig ib cr cg cb))
                           ;; clipping haut et bas
                           (when (or (< h 0) (>= h *screen-height*)) (return-fro
m line_hzg))
                           ;; erreur de sens
                           (when (> xd xf) (return-from line_hzg))

                           (when (/= xd xf)
                             (setf deltaz (when-delta1 zf zd xf xd)
                                   ir (when-delta1 rf rd xf xd)
                                   ig (when-delta1 gf gd xf xd)
                                   ib (when-delta1 bf bd xf xd)))
```

```lisp
                         (setf curz (ash zd 8))
                         ;; clipping droit
                         (when (>= xf *screen-width*) (setf xf (- *screen-widt
h* 1)))
                         (do ((i xd (1+ i)))
                             ((> i xf))
                           (declare (type int32 i))
                           (if (>= i 0) (setpixz i h (ash curz -8)
                                                     (the card8 (ash cr -8))
                                                     (the card8 (ash cg -8))
                                                     (the card8 (ash cb -8))))
                           (incf curz deltaz)
                           (incf cr ir)
                           (incf cg ig)
                           (incf cb ib)))))

          (let ((delta1 0)
                (delta2 0)
                (deltaz1 0)
                (deltaz2 0)
                (d1r 0) (d1g 0) (d1b 0) ; delta1 color
                (d2r 0) (d2g 0) (d2b 0) ; delta2 color
                (rd 0) (gd 0) (bd 0)    ; debut color
                (rf 0) (gf 0) (bf 0)    ; fin color
                (xd 0)
                (xf 0)
                (zd 0)
                (zf 0))

            (declare (type int32 delta1 delta2
                           deltaz1 deltaz2
                           d1r d1g d1b
                           d2r d2g d2b
                           rd gd bd
                           rf gf bf
                           xd xf zd zf))

            ;; trie des y
            (abc-swap X Y R G B Z)

            ;; remplissage
            (when (/= ya yc)
              (setf delta1 (the int32 (when-delta1 xa xc ya yc))
                    deltaz1 (the int32 (when-delta1 za zc ya yc))
                    d1r (when-d ra rc ya yc)
                    d1g (when-d ga gc ya yc)
                    d1b (when-d ba bc ya yc)))

            (when (/= ya yb)
              (setf delta2 (the int32 (when-delta1 xb xa yb ya))
                    deltaz2 (the int32 (when-delta1 zb za yb ya))
                    d2r (when-d rb ra yb ya)
                    d2g (when-d gb ga yb ya)
                    d2b (when-d bb ba yb ya)))

            (setf xd (ash (the int32 xa) 8))
            (setf zd (ash (the int32 za) 8))
            (setf rd (ash ra 8))
            (setf gd (ash ga 8))
            (setf bd (ash ba 8))
            (list-match-setf (xf zf rf gf bf) (xd zd rd gd bd))
```



```lisp
            (if-delta 1 2 A B fillpoly3zgour-do-action1 fillpoly3zgour-do-action
2)

            (when (/= yb yc)
              (setf delta2 (the int32 (when-delta1 xb xc yb yc)))
              (setf deltaz2 (the int32 (when-delta1 zb zc yb yc)))
              (setf xf (ash (the int32 xb) 8))
              (setf zf (ash (the int32 zb) 8))
              (setf d2r (when-d rb rc yb yc))
              (setf d2g (when-d gb gc yb yc))
              (setf d2b (when-d bb bc yb yc))
              (setf rf (ash rb 8))
              (setf gf (ash gb 8))
              (setf bf (ash bb 8))

              (if-delta 2 1 B C fillpoly3zgour-do-action1 fillpoly3zgour-do-acti
on2))))))))
```

```
;;; ------------------------------------------------------------------------
;;; $RCSfile: VertexBuffer.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:40 $
;;; $Revision: 1.1.1.1 $
;;; ------------------------------------------------------------------------

(declaim (optimize (speed 3)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 0)))


(deftype rgb    () '(simple-array card8 (3)))
(deftype pcoord () '(simple-array int16 (3))) ;to hold window coordinates


(defun rgb-make (r g b)
  (declare (type card8 r g b))
  (make-array 3 :element-type 'card8 :initial-contents (list r g b)))


;setable accesors
(defmacro rgb-r (col)
  `(aref ,col 0))
(defmacro rgb-g (col)
  `(aref ,col 1))
(defmacro rgb-b (col)
  `(aref ,col 2))


(declaim (inline rgb-set))
(defun rgb-set (col r g b)
  (declare (type rgb col))
  (declare (type card8 r g b))
  (setf (rgb-r col) r
        (rgb-g col) g
        (rgb-b col) b))


(defun pcoord-make-null ()
  (make-array 3 :element-type 'int16))


(defstruct vertex
  (coords   (vec4-make-null)   :type vec4)      ;object coordinates
  (normal   (vec3-make-null)   :type vec3)      ;normal of vertex
  (material (make-material)    :type material)  ;material of vertex
  (color    (rgb-make 0 0 0)   :type rgb)       ;color of vertex
  (tcoords  (vec4-make-null)   :type vec4)      ;world (eye) coordinates
  (tnormal  (vec3-make-null)   :type vec3)      ;normal after modelview transfor
m
  (pcoords  (pcoord-make-null) :type pcoord))   ;window coordinates


(deftype vb-array-index () 'card16)
(defparameter *rgb-white* (rgb-make 255 255 255))
(defparameter *vb-current-color* *rgb-white*)
(defparameter *vb-current-normal* (vec3-make-null))
;;; vbsize must be divisible by 6 to be able to specify complete lines (divisibl
e by 2)
;;; and complete triangles (divisible by 3)
```



```
(defparameter *vb-size* 120)
(defparameter *vb* (make-array *vb-size* :element-type 'vertex))
(defparameter *vb-index* 0)
(defparameter *vb-draw-mode* :UNKNOWN)


(declaim (type (simple-vector *) *vb*))
(declaim (type rgb  *vb-current-color*))
(declaim (type vec3  *vb-current-normal*))
(declaim (type vb-array-index *vb-index* *vb-size*))


;; allocates memory for the vb to avoid consing
(dotimes (i *vb-size*)
  (setf (aref *vb* i) (make-vertex)))


(defmacro set-vb-vertex (i slot-type &rest slot-values)
  "Sets ith vertex slot 'slot-type' to 'slot-values' into the vertex buffer"
  (multiple-value-bind (val1 val2 val3 val4) (list-values slot-values)
    (let* ((4access-list (list 'vec-x 'vec-y 'vec-z 'vec-w))
           (4vals-list (list val1 val2 val3 val4))
           (3access-list (butlast 4access-list))
           (3vals-list (butlast 4vals-list))
           (vertex-name (lambda (x) (symb 'VERTEX- (symbol-name x))))
           (setf-gen (lambda (x y vertex)
                       `(setf (,x (,(funcall vertex-name vertex) (aref *vb* ,i))
) ,y)))
           (build-3-setf (lambda (vertex) (mapcar (lcurry setf-gen nil nil verte
x)
                                                  3access-list 3vals-list)))
           (build-4-setf (lambda (vertex) (mapcar (lcurry setf-gen nil nil verte
x)
                                                  4access-list 4vals-list))))
      (ccase slot-type
        (:material
         `(progn
            (vec4-cpy (material-ambiant (vertex-material (aref *vb* ,i)))
                      (material-ambiant *current-material*))
            (vec4-cpy (material-diffuse (vertex-material (aref *vb* ,i)))
                      (material-diffuse *current-material*))
            (vec4-cpy (material-specular (vertex-material (aref *vb* ,i)))
                      (material-specular *current-material*))
            (setf (material-shininess (vertex-material (aref *vb* ,i)))
                  (material-shininess *current-material*))
            (vec4-cpy (material-emission (vertex-material (aref *vb* ,i)))
                      (material-emission *current-material*))))
        ((:coords :tcoords)
         `(progn ,@(funcall build-4-setf slot-type)))
        ((:pcoords :normal :tnormal :color)
         `(progn ,@(funcall build-3-setf slot-type)))))))


; x,y,z,w in the body are replaced by #{v 0},#{v 1},...
; (with-vec4 v (+ x y z w)) -> (+ #{v 0} #{v 1} #{v 2} #{v 3} )
(defmacro with-vec4 (v &rest body)
  (let* ((x `#{,v 0})
         (y `#{,v 1})
         (z `#{,v 2})
         (w `#{,v 3})
         (newbody (substn (list x 'x y 'y z 'z w 'w) body)))
    `(prog ()
```

```lisp
           ,@newbody)))


; same as with-vec4 with only x,y,z
(defmacro with-vec3 (v &rest body)
  (let* ((x `#{,v 0})
         (y `#{,v 1})
         (z `#{,v 2})
         (newbody (substn (list x 'x y 'y z 'z) body)))
    `(prog ()
           ,@newbody)))


(defun vec4-transform (i)
  "Object to world coordinates transformation"
  (with-vec4 (vertex-coords (aref *vb* i))
        (macrolet ((row-by-col (row)
                              (declare (type (card8) row))
                              `(+ (the single-float (* (the single-float #[*modelv
iew-matrix* ,row 0]) x))
                                  (the single-float (* (the single-float #[*modelv
iew-matrix* ,row 1]) y))
                                  (the single-float (* (the single-float #[*modelv
iew-matrix* ,row 2]) z))
                                  (the single-float (* (the single-float #[*modelv
iew-matrix* ,row 3]) w)))))
        (set-vb-vertex i :tcoords
                       (row-by-col 0)
                       (row-by-col 1)
                       (row-by-col 2)
                       (row-by-col 3)))))


(defun normal-transform (i)
  "Transform normals by the upper-left 3x3 matrix of the modelview matrix for lighting"
  (with-vec3 (vertex-normal (aref *vb* i))
        (macrolet ((row-by-col (row)
                                 (declare (type (card8) row))
                                 `(+ (the single-float (* (the single-float #[*mod
elview-matrix* ,row 0]) x))
                                     (the single-float (* (the single-float #[*mod
elview-matrix* ,row 1]) y))
                                     (the single-float (* (the single-float #[*mod
elview-matrix* ,row 2]) z)))))
          (set-vb-vertex i :tnormal
                         (row-by-col 0)
                         (row-by-col 1)
                         (row-by-col 2)))))


(defun vec4-project (i)
  "World to window coordinates transformation"
  (with-vec4 (vertex-tcoords (aref *vb* i))
             (macrolet ((row-by-col (row)
                                      (declare (type card8 row))
                                      `(+ (the single-float (* (the single-float #
[*projection-matrix* ,row 0]) x))
                                          (the single-float (* (the single-float #
[*projection-matrix* ,row 1]) y))
                                          (the single-float (* (the single-float #
[*projection-matrix* ,row 2]) z))
                                          (the single-float (* (the single-float #
```



```lisp
[*projection-matrix* ,row 3]) w)))))
               ;xc,yc,zc,wc: clip coordinates of the transformed vertex
               (let ((xc (row-by-col 0))
                     (yc (row-by-col 1))
                     (zc (row-by-col 2))
                     (wc (row-by-col 3)))
                 (declare (type single-float xc yc zc wc))
                 ;perspective division (by wc=-z) and viewport transformation fo
r final window coordinates
                 (set-vb-vertex i :pcoords
                    (+ (the card16 (truncate (the single-float (* *viewport-half
-width*  (/ xc wc))))) *viewport-centerx*)
                    (+ (the card16 (truncate (the single-float (* *viewport-half
-height* (/ yc wc))))) *viewport-centery*)
                    (the card16 (truncate (the single-float zc))))))))


(defun do-transforms ()
  (do ((i 0 (1+ i)))
      ((>= i *vb-index*))
    (declare (type vb-array-index i))
    (vec4-transform i)
    (normal-transform i)
    (vec4-project i)))


(let ((sum (vec3-make-null)))
  (declare (type vec3 sum))
  (defun do-lighting ()
    (do ((i 0 (1+ i))
         (vbi nil))
        ((>= i *vb-index*))
      (declare (type vb-array-index i))
      (setf vbi (aref *vb* i))
      (do ((j 0 (1+ j))
           (r 0.0)
           (g 0.0)
           (b 0.0)
           (vnormal    (vertex-tnormal (aref *vb* i)))
           (mambiant   (material-ambiant   (vertex-material vbi)))
           (mdiffuse   (material-diffuse   (vertex-material vbi)))
           (mspecular  (material-specular  (vertex-material vbi)))
           (mshininess (material-shininess (vertex-material vbi)))
           (memission  (material-emission  (vertex-material vbi))))
          ((>= j *nblight*)
           (set-vb-vertex i :color
                          (min (the int16 (truncate (the (single-float #.(float
most-negative-fixnum)
                                                                        #.(float
most-positive-fixnum)) (* r 255)))) 255)
                          (min (the int16 (truncate (the (single-float #.(float
most-negative-fixnum)
                                                                        #.(float
most-positive-fixnum)) (* g 255)))) 255)
                          (min (the int16 (truncate (the (single-float #.(float
most-negative-fixnum)
                                                                        #.(float
most-positive-fixnum)) (* b 255)))) 255)))

        (declare (type card8 j)
                 (type (single-float #.(float most-negative-fixnum)
                                     #.(float most-positive-fixnum)) r g b)
```

```
                 (type vec3 vnormal)
                 (type vec4 mambiant mdiffuse mspecular memission)
                 (type single-float mshininess))
        (if (aref *tlightsbool* j)
            (let* ((lposition  (light-position (aref *tlights* j)))
                   (lambiant   (light-ambiant  (aref *tlights* j)))
                   (ldiffuse   (light-diffuse  (aref *tlights* j)))
                   (lspecular  (light-specular (aref *tlights* j)))
                   (cos-angle1 0.0)
                   (cos-angle2 0.0))
              (declare (type vec4 lposition lambiant ldiffuse lspecular)
                       (type single-float cos-angle1 cos-angle2))
              ;; ambiant section
              (incf r (+ (* #{ lambiant 0 } #{ mambiant 0 }) #{ memission 0 }))
              (incf g (+ (* #{ lambiant 1 } #{ mambiant 1 }) #{ memission 1 }))
              (incf b (+ (* #{ lambiant 2 } #{ mambiant 2 }) #{ memission 2 }))
              ;; diffuse section
              (setf cos-angle1 (vec-dot-product vnormal lposition))
              (if (> cos-angle1 0.0)
                  (progn
                    (incf r (* (* #{ mdiffuse 0 } #{ ldiffuse 0 }) cos-angle1))
                    (incf g (* (* #{ mdiffuse 1 } #{ ldiffuse 1 }) cos-angle1))
                    (incf b (* (* #{ mdiffuse 2 } #{ ldiffuse 2 }) cos-angle1)))
)
              ;; specular section
              (vec3-set sum #{lposition 0} #{lposition 1} (+ #{lposition 2} 1.0)
)
              (vec-set-normalize sum)
              (setf cos-angle1 (vec-dot-product vnormal sum))
              (when (> cos-angle1 0.0)
                (setf cos-angle2 (the single-float (expt (the (single-float 0.0)
 cos-angle1)
                                                         (the single-float  mshi
niness))))
                (incf r (* (* #{ mspecular 0 } #{ lspecular 0 }) cos-angle2))
                (incf g (* (* #{ mspecular 1 } #{ lspecular 1 }) cos-angle2))
                (incf b (* (* #{ mspecular 2 } #{ lspecular 2 }) cos-angle2)))))
))))


(defun do-normals-normalize ()
  (do ((i 0 (1+ i)))
      ((>= i *vb-index*))
    (vec-set-normalize (vertex-tnormal (aref *vb* i)))))


(defun draw-points ()
  (do ((i 0 (setf i (+ i 1))))
      ((>= i *vb-index*))
    (declare (type vb-array-index i ))
    (let ((x (aref (vertex-pcoords (aref *vb* i)) 0))
          (y (aref (vertex-pcoords (aref *vb* i)) 1))
          (r  (aref (vertex-color (aref *vb* i)) 0))
          (g  (aref (vertex-color (aref *vb* i)) 1))
          (b  (aref (vertex-color (aref *vb* i)) 2)))
      (setpix x y r g b))))


(defun draw-lines ()
  (do ((i 0 (setf i (+ i 2))))
      ((>= i *vb-index*))
    (declare (type vb-array-index i ))
```



```
    (let ((x1 (aref (vertex-pcoords (aref *vb* i)) 0))
          (y1 (aref (vertex-pcoords (aref *vb* i)) 1))
          (x2 (aref (vertex-pcoords (aref *vb* (+ i 1))) 0))
          (y2 (aref (vertex-pcoords (aref *vb* (+ i 1))) 1))
          (r  (aref (vertex-color (aref *vb* i)) 0))
          (g  (aref (vertex-color (aref *vb* i)) 1))
          (b  (aref (vertex-color (aref *vb* i)) 2)))
      (rgbline x1 y1 x2 y2 r g b))))


(defun draw-line-loop ()
  (draw-line-strip)
  ;; last line = last to first point
  (let ((lastline (1- *vb-index*)))
    (rgbline
     (aref (vertex-pcoords (aref *vb* lastline)) 0)
     (aref (vertex-pcoords (aref *vb* lastline)) 1)
     (aref (vertex-pcoords (aref *vb* 0)) 0)
     (aref (vertex-pcoords (aref *vb* 0)) 1)
     (aref (vertex-color (aref *vb* lastline)) 0)
     (aref (vertex-color (aref *vb* lastline)) 1)
     (aref (vertex-color (aref *vb* lastline)) 2))))


(defun draw-line-strip ()
  (do ((i 0 (incf i)))
      ((>= i (1- *vb-index*)))
    (declare (type vb-array-index i ))
    (let ((x1 (aref (vertex-pcoords (aref *vb* i)) 0))
          (y1 (aref (vertex-pcoords (aref *vb* i)) 1))
          (x2 (aref (vertex-pcoords (aref *vb* (+ i 1))) 0))
          (y2 (aref (vertex-pcoords (aref *vb* (+ i 1))) 1))
          (r  (aref (vertex-color (aref *vb* i)) 0))
          (g  (aref (vertex-color (aref *vb* i)) 1))
          (b  (aref (vertex-color (aref *vb* i)) 2)))
      (rgbline x1 y1 x2 y2 r g b))))


(declaim (inline visible-triangle?))
(defun visible-triangle? (x1 y1 x2 y2 x3 y3)
  (declare (type int16 x1 y1 x2 y2 x3 y3))
  (< (- (* (the int16 (- y2 y1))
           (the int16 (- x3 x1)))
        (* (the int16 (- y3 y1))
           (the int16 (- x2 x1)))) 0))


(defmacro draw-triangle (triangle-type)
  (let ((vertex1
          (case triangle-type
            ((simple strip) 'i)
            (fan    '0)))
        (vertex2
          (case triangle-type
            ((simple strip) '(+ i 1))
            (fan    'i)))
        (vertex3
          (case triangle-type
            ((simple strip) '(+ i 2))
            (fan    '(+ i 1))))
        (test
          (case triangle-type
```

```
          (simple '(((i 0 (incf i 3)))
                    ((>= i *vb-index*))))
          (strip  '(((i 0 (incf i))
                     (inversevertexe 'nil (setf inversevertexe (not inversevert
exe))))
                    ((>= i (- *vb-index* 2)))))
          (fan    '(((i 1 (incf i)))
                    ((>= i (- *vb-index* 1)))))))
       (body
        (case triangle-type
          ((simple fan) '((if (visible-triangle? x1 y1 x2 y2 x3 y3)
                             (if (eq *shade-model* :FLAT)
                                 (if *use-zbuffer*
                                     (fillpoly3z x1 y1 z1
                                                 x2 y2 z2
                                                 x3 y3 z3
                                                 r1 g1 b1)
                                   (fillpoly3 x1 y1
                                              x2 y2
                                              x3 y3
                                              r1 g1 b1))
                               (if *use-zbuffer*
                                   (fillpoly3zgour x1 y1 z1 r1 g1 b1
                                                   x2 y2 z2 r2 g2 b2
                                                   x3 y3 z3 r3 g3 b3)
                                 (fillpoly3gour x1 y1 r1 g1 b1
                                                x2 y2 r2 g2 b2
                                                x3 y3 r3 g3 b3))))))
          (strip        '((if (and  (not inversevertexe) (visible-triangle? x1
y1 x2 y2 x3 y3))
                             (if (eq *shade-model* :FLAT)
                                 (if *use-zbuffer*
                                     (fillpoly3zgour x1 y1 z1 r1 g1 b1
                                                     x2 y2 z2 r2 g2 b2
                                                     x3 y3 z3 r3 g3 b3)
                                   (fillpoly3gour x1 y1 r1 g1 b1
                                                  x2 y2 r2 g2 b2
                                                  x3 y3 r3 g3 b3))))

                          (if (and inversevertexe (visible-triangle? x2 y2 x1 y
1 x3 y3))
                              (if (eq *shade-model* :FLAT)
                                  (if *use-zbuffer*
                                      (fillpoly3zgour x2 y2 z2 r2 g2 b2
                                                      x1 y1 z1 r1 g1 b1
                                                      x3 y3 z3 r3 g3 b3)
                                    (fillpoly3gour x2 y2 r2 g2 b2
                                                   x1 y1 r1 g1 b1
                                                   x3 y3 r3 g3 b3)))))))))
    `(do ,@test
        (declare (type vb-array-index i ))
      (let ((x1 (aref (vertex-pcoords (aref *vb* ,vertex1)) 0))
            (y1 (aref (vertex-pcoords (aref *vb* ,vertex1)) 1))
            (z1 (aref (vertex-pcoords (aref *vb* ,vertex1)) 2))

            (x2 (aref (vertex-pcoords (aref *vb* ,vertex2)) 0))
            (y2 (aref (vertex-pcoords (aref *vb* ,vertex2)) 1))
            (z2 (aref (vertex-pcoords (aref *vb* ,vertex2)) 2))

            (x3 (aref (vertex-pcoords (aref *vb* ,vertex3)) 0))
            (y3 (aref (vertex-pcoords (aref *vb* ,vertex3)) 1))
            (z3 (aref (vertex-pcoords (aref *vb* ,vertex3)) 2))
```



```
            (r1  (aref (vertex-color (aref *vb* ,vertex1)) 0))
            (g1  (aref (vertex-color (aref *vb* ,vertex1)) 1))
            (b1  (aref (vertex-color (aref *vb* ,vertex1)) 2))

            (r2  (aref (vertex-color (aref *vb* ,vertex2)) 0))
            (g2  (aref (vertex-color (aref *vb* ,vertex2)) 1))
            (b2  (aref (vertex-color (aref *vb* ,vertex2)) 2))

            (r3  (aref (vertex-color (aref *vb* ,vertex3)) 0))
            (g3  (aref (vertex-color (aref *vb* ,vertex3)) 1))
            (b3  (aref (vertex-color (aref *vb* ,vertex3)) 2)))

        ,@body))))


(defun draw-triangles ()
  (draw-triangle simple))


(defun draw-triangle-strip ()
  (draw-triangle strip))


(defun draw-triangle-fan ()
  (draw-triangle fan))


(declaim (inline clear-vb))
(defun clear-vb ()
  "Clear the vb to some default values and put the index at the beginning"
  (setf *vb-index* 0))


;(declaim (inline Color))
(defun Color (r g b)
  (declare (type card8 r g b))
  (rgb-set *vb-current-color* r g b)
  (vec4-set (material-diffuse *current-material*)
            (/ r 255.0)
            (/ g 255.0)
            (/ b 255.0)
            1.0))


(defconstant +begin-drawing-keys+ '(:POINTS :LINES :LINE-STRIP :LINE-LOOP :TRIANG
LES :TRIANGLE-STRIP :TRIANGLE-FAN))
(defconstant +begin-drawing-funcs+ '(draw-points draw-lines draw-line-strip draw-
line-loop draw-triangles draw-triangle-strip
                                                  draw-triangle-fan))


(defmacro Begin (mode &rest body)
  "Begin block holds rendering primitives"
  ;first, check if mode is valid
  (when (not (member mode +begin-drawing-keys+))
    (error "Begin: unknown primitive type: ~S" mode))

  (let ((bad-form nil))
    (labels ((body-form-valid-p (form)
                                (case (car form)
                                  ((Color Normal Vertex ArrayElement declare) t)
```

```
                                   ((if when dotimes let let*) (body-valid-p (cdd
r form)))
                                   ((do) (body-valid-p (cdddr form)))
                                   (t (progn (setf bad-form form)
                                             nil))))
             (body-valid-p (body)
                               (if (null body) t
                                 (and (body-form-valid-p (car body)) (body-valid-p (
cdr body))))))
       ;second, check for unwanted function calls in the body
       (when (not (body-valid-p body))
         (error "Begin: illegal primitive call in form: ~S" bad-form))
       `(labels ((flush-vb ()
                          (do-transforms)
                          (when *do-normals-normalize* (do-normals-normalize))
                          (when *do-lighting* (do-lighting))
                          ;find the appropriate drawing function based on mode
                          ,(list (elt +begin-drawing-funcs+ (position mode +begi
n-drawing-keys+))))

                (Normal (nx ny nz)
                        (declare (type single-float nx ny nz))
                        (vec3-set *vb-current-normal* nx ny nz)
                        t)

                (Vertex (x y z w)
                        (declare (type single-float x y z w))

                                       ;if the vb is full, flush it
                        (when (>= *vb-index* *vb-size*)
                          (progn (flush-vb)
                                 (clear-vb)))
                                       ;accumulate the vertex at the end of the
 vb
                        (set-vb-vertex *vb-index* :coords x y z w)
                        (set-vb-vertex *vb-index* :normal
                                       #{*vb-current-normal* 0} #{*vb-current-no
rmal* 1} #{*vb-current-normal* 2})
                        (set-vb-vertex *vb-index* :material *current-material*)
                        (set-vb-vertex *vb-index* :color (rgb-r *vb-current-colo
r*)
                                       (rgb-g *vb-current-color*)
                                       (rgb-b *vb-current-color*))
                        (incf *vb-index*)))

          (declare (inline Normal))
          (clear-vb)
          ,@body
          (flush-vb)))))


;;; Vertex arrays stuff

(defparameter *vertex-array* nil)
;(declaim (type (simple-vector *) *vertex-array*))


(defmacro VertexPointer (array)
  `(setf *vertex-array* ,array))


(defmacro ArrayElement (i)
```



```
  (let ((index (gensym)))
    `(let ((,index (* 4 (the card32 ,i))))
       (declare (type card32 ,index))
       (Vertex (aref (the simple-array (eval *vertex-array*)) ,index)
               (aref (the simple-array (eval *vertex-array*)) (incf ,index))
               (aref (the simple-array (eval *vertex-array*)) (incf ,index))
               (aref (the simple-array (eval *vertex-array*)) (incf ,index))))))


(defmacro DrawElements (mode count indices)
  `(Begin ,mode
          (dotimes (i ,count)
            (declare (type (simple-array card16 (*)) ,indices))
            (ArrayElement (aref ,indices i)))))


(defmacro DrawArrays (mode first count)
  `(Begin ,mode
          (dotimes (i ,count)
            (ArrayElement (+ ,first i)))))
```