Università degli Studi di Roma
"Tor Vergata"
Facoltà di Ingegneria
Corso di laurea in Ing. Informatica

Relazione finale
Creazione e animazione interattiva di grafica tridimensionale

Relatore

Francesco Martinelli

Candidato

Michele Martone

Anno accademico 2003/2004

last revision 2004.11.05, in LaTeX

Contents

I Overview **8**

1 Introduction **9**

2 Thesis Objectives Overview **12**

3 Thesis Objectives Detail **14**
3.1 Terrain representation 14
3.1.1 Terrain representation in *UGP* 14
3.2 Remote controlled manipulator visualization 15
3.2.1 Manipulator visualization in *UGP* 16
3.3 Digital imaging and computer graphics in medical environments 16
3.3.1 A needle-like object penetrating a soft tissue in *UGP* . . 18

II Theoretical foundations **20**

4 Two dimensional graphics **21**
4.1 Computer colour and light 21
4.1.1 Colour spaces 22
4.1.2 RGB colour space 22
4.2 Transformations and notation 23
4.3 Punctual transformations 24
4.3.1 Desaturation 24
4.3.2 Negative 24
4.3.3 Saturation 24
4.3.4 Brightness and Vividness 25
4.3.5 Whiteness 25
4.3.6 Blackness 25
4.4 Non Punctual transformations 26
4.4.1 Blur and matrix convolutions 26
4.4.2 Contrast 28
4.4.3 Despeckle 29
4.4.4 Sharpening 30
4.5 Resizing (resampling) 31
4.5.1 Resizing - Upsampling 31
4.5.2 Resizing - Downsampling 32

5 Three dimensional graphics **33**
5.1 Points, planes, and straight lines 33
5.1.1 Containment of points in polygons 34
5.1.2 Ray - polygon intersection 35
5.1.3 Back and front polygons 35
5.2 Meshes 36
5.2.1 Hull properties 36
5.2.2 Some properties of triangle meshes 37
5.2.3 Convexity 38
5.2.4 The implemented data structure 39

III Software architecture **41**

6 Introduction **42**
6.1 Programming style 42
6.2 *UGP* components 43
6.2.1 *UGP* components by dissection 43
6.2.2 Minor components and accessory classes 44
6.2.3 The components and libraries 45
6.3 Code documentation 45

7 Engine and control **47**
7.1 The base engine class and inheritance 47
7.2 The derived class SDLengine 47
7.2.1 OpenGL initialization 48
7.2.2 Input handling 48
7.3 The command console and control 49

8 The rendering process **51**
8.1 Camera 51
8.1.1 The camera and the viewing system 51
8.1.2 World to view space transformation 52
8.1.3 Perspective transformations and 3D screen space 53
8.1.4 Transformations and OpenGL 54
8.1.5 Two camera classes 54
8.1.6 Communication with the engine 55
8.2 2D Drawer 56
8.2.1 Text drawing 57
8.3 3D Drawer and geometry drawing 58
8.3.1 OpenGL initialization and portability 58
8.3.2 Setting up OpenGL state 58
8.3.3 OpenGL drawing commands 59

9 Entities **61**
9.1 *Dez3DEntity*, a common base class for entities 61
9.1.1 Entity flags 61
9.1.2 Entity geometry and text drawing 62
9.1.3 Entity update and input 62
9.2 Some examples with pictures 62

9.2.1 Menger sponge 63
9.2.2 Bouncing sphere 63
9.2.3 Shootable 64
9.2.4 Tissue 64
9.2.5 The robot arm 65
9.2.6 2D Image Displayer 66

10 *MeshCreator* class as a shape factory **69**
10.1 A primitive shape creator 69
10.2 The *MeshCreator* command string 69

11 *UGP* usage **71**
11.1 Mode switching commands 71
11.2 Normal mode commands 71
11.3 Console mode commands 72

List of Figures

1.1 A screenshot of *UGP*. 9
1.2 Western Australia University Telelabs Project. 10

2.1 *UGP* in terrain exploration mode, in wireframe. 13

3.1 *UGP* in terrain exploration mode, textured. 15
3.2 Screenshot of *UGP* with the controlled manipulator. 17
3.3 *UGP* simulating an elastic, breakable tissue. 19

4.1 Wassily Kandinsky : Improvisation 21a. 1911. Oil on canvas, 96x105 cm. Muenchen, Staedtische Galerie in Lenbach, Germany. 21
4.2 A colour construction showing the colour cube in RGB space. . 23
4.3 Negative. 25
4.4 Whiteness. 26
4.5 Blackness. 27
4.6 The Tartaglia-Pascal triangle. 28
4.7 Normalized binomial coefficient values for the correspondent kernel. 29
4.8 Contrast. 30

5.1 A point belonging to a triangle. 34
5.2 Triangle orientation. 36
5.3 Incorrectly (a), and correctly (b) ordered edges. 37
5.4 An edge. 37
5.5 A triangle. 37
5.6 A triangle enclosed by three edges. 38
5.7 Edges sharing in adjacent triangles. 38
5.8 A tetrahedron with split faces. 39
5.9 A convex and a non convex mesh. 39
5.10 An example of mesh representation. 40

6.1 The main component classes of *UGP*. 43
6.2 Accessory classes. 44

8.1 World and view coordinate systems. 52
8.2 The *view frustum* enclosed by the *far* and *view* plane. 53
8.3 Spherical coordinate system. 55
8.4 OpenGL API hierarchy in Unix and Windows *(from www.sgi.com)*. 58

9.1 *DezMenger* objects, representing Menger sponges of order 1,2,3, and through the axial 'hole' of an order 1 one. 63
9.2 A bouncing ball. 64
9.3 A *DezShootable* class object : to be selected, selected, hit, missed. 65
9.4 Our sample image and its histogram. 66
9.5 Blur and edge revealing operations. 67
9.6 Resizing of images in *UGP*. 67
9.7 Contrast enhancing and the concatenation of partial desaturation,contrast and saturation. 67
9.8 The sample image after box pixelating and negative effects. . . . 68

10.1 Structure of pointer interconnections for a tetrahedron. 69

Part I

Overview

Chapter 1

Introduction

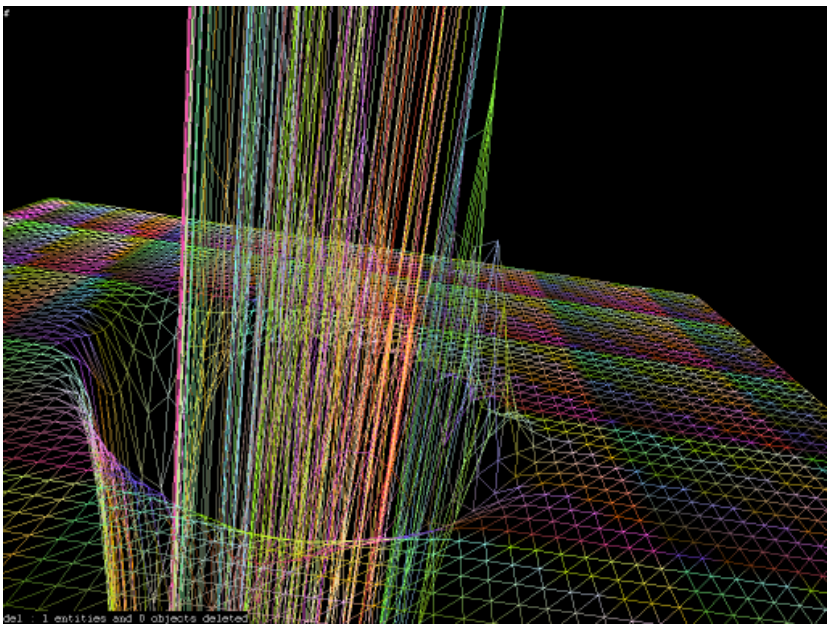

Figure 1.1: A screenshot of *UGP*.

As electronics miniaturization progress proceeds, the increased capabilities of computing machines give opportunity to new application fields. One of these is the vast field of activities related to computer graphics. Computer graphics is concerned with creation, description (internal representation), manipulation and visualization of graphical objects. Usually each of these activities is performed using a different software tool.

Creation is the process of describing graphical objects using a GUI or a script language, and produces some structured data which usually is permanently storable.

A description of the graphical object is needed to let the programs loading it into memory and perform some further editing and/or visualization. Different

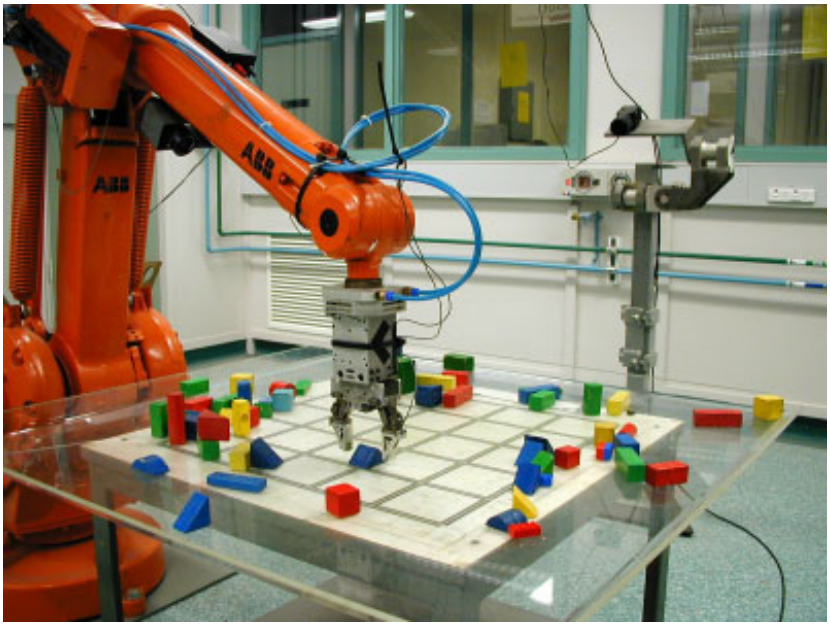

Figure 1.2: Western Australia University Telelabs Project.

representations are possible for the same object, and many have evolved to suite certain needs.

For example, while an implicit description of an object (imagine, for a sphere: $\left\{p \quad | \quad p \in \mathbb{R}^3, \|p - C\| = R, for \quad some \quad R \in \mathbb{R}, C \in \mathbb{R}^3\right\}$) is usually elegant and easily readable to us, it may not be the best for a quick on-screen rendering, since the system should be able to compute and visualize all points in P (an infinite cardinality set) in a reasonable amount of time.

An explicit representation of the object, on the other hand, could be hard to understand for the user (an example of explicit representation could be an enumeration of facets of a polyhedron), but easier to handle by the system. The need of a trade-off between ease of representation and computation (or even other parameters) arises.

In fact, among the existing file formats, there is a clear differentiation between those oriented towards visualization and those oriented towards editing, or even those oriented towards elegant representation. Manipulation too is affected by the internal representation on graphical objects. Imagine a shape represented by a big number of polygonal facets in an interactive editing program: without a coexistent, easier to manipulate alternative representation, in order to expand the object the user should select all of them, and drag them one by one.

A smart facility here would be a structure holding logical information about the polyhedron (with its constraints, like regularity, convexity, etc...).

In practice, the aforementioned trade-off is reached with redundancy in the representation. Redundant structures hold more information than necessary, usually providing facilities to certain operations. Once the graphical objects are ready, they can be drawn on-screen in the final application context (visualiza-

tion). Depending on the chosen representation, the visualization could or could not require further computation to be performed on the stored data.

Normally, the more suitable choice is made with reference to the context. For example, in a visual editing program, the computation speed will be sacrificed for the ease of interactive manipulation. Instead, in an interactive virtual reality tool, usually all the focus will be put on the rendering speed, and possibly to the ease of certain fixed operations (collisions or certain actions, like opening of doors).

In the second example the needs are mixed, and often a part of the graphical 'world' is stored 'statically' (no interactions possible), and another could be marked as 'dynamic' (physical or logical interactions possible, like holding, moving, deformations).

This leads to differentiated internal representation, and thus increased implementation complexity.

In facts, computer graphics is a relatively new field of research and more and more algorithms are being developed for its various problems. The disciplines involved, or potentially involved, besides computer science, are algebra, geometry, computational geometry, graph theory, physics, and optics.

Chapter 2

Thesis Objectives Overview

The purpose of the work done during the training period was the creation of a basic graphics software library, devoted to generation, manipulation and visualization of two dimensional (raster) and three dimensional (vectorial) graphics.

As thesis work, the implementation of the library in example software, one for batch 2D image processing, *UGP*, one for demonstration of combined 2D/3D capabilities. The former, following the UNIX style, is command line driven and implements : image loading, manipulation (some punctual and non punctual filters, lines drawing) and saving.

The latter is a bit more multipurpose, since it is initialized by a script file which can set up the environment as desired. It basically consists in a camera flying around in a world populated by graphical objects, some of them animated and interactive.

The camera speed is real time, thus independent of the machine speed.

Interaction with the world is possible thanks to the command console built in the program.

This allows the user to communicate in a flexible manner (through 'messages') with so-called 'entities' (the interactive elements in the world) and the program parameters (drawing settings, navigation options) [1].

From the command line it is also possible to spawn new entities or meshes (just graphical objects), or delete them.

Bypassing the command line, it is even possible to take interactive control of entities (sort of 'possessing' them) through controlling them with direct input (pointer and keyboard). With this degree of flexibility it was possible to create some examples of practical application of computer graphics.

These are:

3.1 Terrain representation and navigation

3.2 Remote controlled manipulator visualization

3.3 A needle-like object penetrating a soft tissue

[1] Described in 9.1.1.

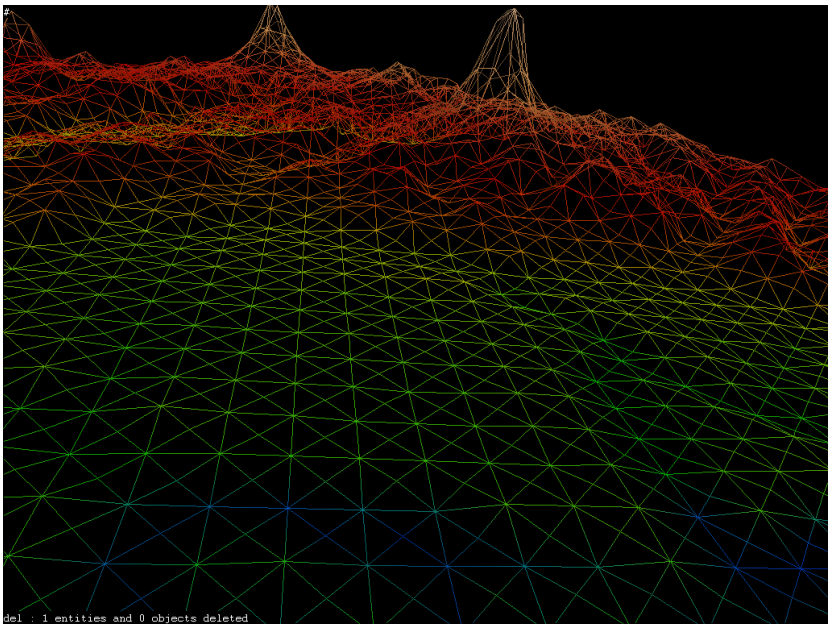

Figure 2.1: *UGP* in terrain exploration mode, in wireframe.

Chapter 3

Thesis Objectives Detail

3.1 Terrain representation

In applications involving remote robotics, meteorology, environmental sciences, or flight simulators, it is often desired the representation of terrain related data, like earth landscapes, altimetries, or geological maps.

With such a vast range of applications and specifications, and different software requirements, implemented techniques may vary for each project. Thus, for example, the best choice for a flight simulator (real time rendering, scalable quality), may not be the best for a landscape generator (statical rendering, fixed quality).

Usually terrain representation involves a great amount of raw data to be stored, indexed, processed, and finally, rendered. Problems often arise when dealing with unacceptably high amounts of data. This data usually comes from geological or satellite surveys, and needs further processing to be visualizable.

Examples of such processing are normalization of data values in certain ranges, reconstruction of missing areas, smoothing and statistical correction of eventual incorrect data. Since the aforementioned procedures don't involve neither graphical computation, nor visualization, they are not directly a rendering trouble. But, in order to be suitable for visualization, collected data should be properly organized into structures optimized for the rendering technique.

For example : in a navigation tool, like a flight simulator, there is the need of showing on screen only a small portion of the world surrounding the camera, and avoiding unnecessary computation of the non-encompassing areas. These areas will be drawn when the aircraft will be in proximity of them. There exists vast literature offering a lot of techniques for this range of problems, and vast research is ongoing too.

3.1.1 Terrain representation in *UGP*

In this application there is a limited possibility to experiment with 3D terrain navigation, as the program is capable of generating a terrain mesh based on height and colour descriptions.

The terrain is generated as a triangle mesh by a software component, of

MeshCreator class[1].

The information about mesh geometry is contained in an image file, in *BMP* format[2]. This contains the heights the vertices should have. The forementioned *BMP* file is referred as a *heightmap*. The colour values of the vertices (and triangles) are taken from another *BMP* file, preferrably of the same size (but not necessarily), a *colormap*.

The topology of the object is generated automatically as a planar mesh of adjacent triangles. Each vertex is adjacent to a variable number of other vertices, depending to its position.

The following command:

```
meshcreator terrain heightmap data/gcHMsmall.bmp-
 colormap data/gcCMsmall.bmp xCells 32 yCells 32-
 bounded 0 0 0 100 100 20-
 texture data/textures/grandCanyon256x256.bmp
```

tells *UGP* to create a *terrain mesh*, of 32×32*cells*, bounded in a $100 \times 100 \times 20$ box. The height values will be taken from *data/gcHMsmall.bmp*, while the colour values from *data/gcCMsmall.bmp*. Additionally, a texture can be specified, in this case from the *data/textures/grandCanyon256x256.bmp* file.

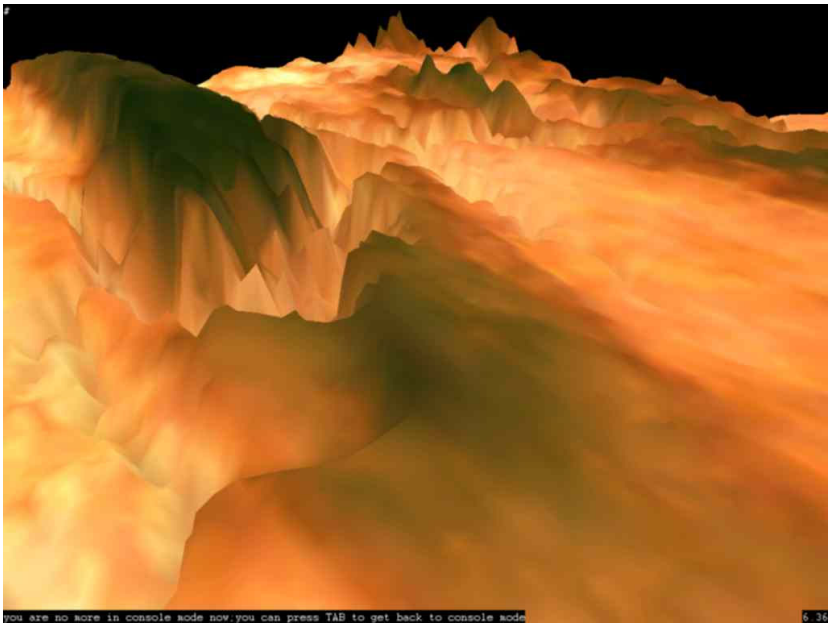

Figure 3.1: *UGP* in terrain exploration mode, textured.

3.2 Remote controlled manipulator visualization

Another area where visualization can come in hand is remote visual monitoring of manipulators. Monitoring could be done with a camera pointed on the

[1]Further description of Mesh class in 5.2 and 10.1.
[2]In 24 bits per pixel mode.

manipulator, and sending frames from time to time to the monitoring station. However, the shortcoming of this choice is the high need for bandwith of the video stream data. Another choice could use the position encoders placed on the manipulator to extrapolate its configuration.

Its constructive parameters (each link's length, angles, and so on) could be used to compile an initial Denavit-Hartenberg table, and with actual data about angles between links and extensions, the current configuration of the manipulator would be set. For example, one can imagine a battery of networked manipulators. Each one, at start-up time, would send its topological and geometrical parameters (e.g.: a Denavit-Hartenberg table, and possibly some information about its arms shape) through the network to the station. This could use this initial description, in combination with dynamical data sent during operation time, to render on-screen the robot in the current configuration. The visualization of a robot arm would thus be an inexpensive way to enhance robot control by human supervisors.

Such application is useful in telerobotics, a field born in the 1960's, in the NASA laboratories. Nowadays telerobotics is taking growing advantage of graphical capabilities of modern computing machines, and a lot of projects is going on or have been done.

For example, in the University of Western Australia (http://telerobot.mech.-uwa.edu.au/) a robot has been interfaced to the Internet through a web interface, in a manner that anyone could control the robot for a certain time period, and see the manipulator through a webcam. Another similar projects are taking place (http://www.robotic.dlr.de/VRML/Rotex/) , with visualization through VRML.

3.2.1 Manipulator visualization in *UGP*

For demonstrative sake, this application is capable of rendering on screen a multi-link manipulator, described through information stored in a text file [3]. This file should contain information, written in plain text, about the number of the links and their initial reciprocal position, and possibly the colour and the shape of the links (prisms, cylinders, cones, sphere).

Due to the interactive nature of the program and the absence of manipulators in the house of the author, the link parameters can be adjusted dynamically by the user. In this implementation, the manipulator is an entity, and thus controllable by the user.

3.3 Digital imaging and computer graphics in medical environments

The use of digital imaging in medical environment began to grow in the 1970's, when one by one, every major medical imaging modality finally could be digitized [ZONNEV]. A milestones is 'digital vascular imaging' (1979).

A common problem in radiography was incorrect exposure, solved with devices digitizing the image and producing the correct contrast, by systems as the PCR (1982), or Thoravision. Computed tomography, originally developed for head scanning, produced cross-sections of the brain tissue, on photographs.

[3] More detailed informations in section 9.2.5.

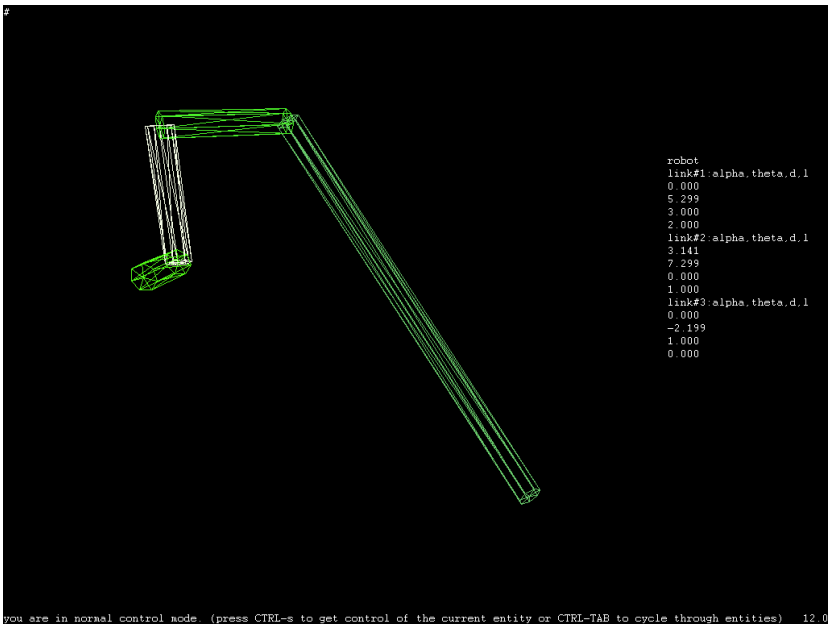

Figure 3.2: Screenshot of *UGP* with the controlled manipulator.

Now it is possible to send the acquired and digitized data from the front-end (the CT scanner) to the back-end (the operator console), for immediate processing, visualization and highlighting. The lower consumption of modern X-ray generators allow the use of CT even in portable ways, thus allowing emergency uses. Magnetic Resonance Imaging (MRI), due to rapid data acquisition and real-time capabilities is gaining popularity over CT. Techniques as FLAIR (Fluid Attenuation Inversion Recovery) allow further contrast increase for the tissues of interest. Dynamic techniques allow the monitoring of temperature (e.g.:cryotherapy, hyperthermia, focused ultrasound), or quantification of blood velocity.

Also ultrasound is gaining benefits from digital data representation, in techniques as colour velocity imaging (CVI) and CVI-Q. With new contrast media development, with tiny gas bubbles generating increased frequency ultrasounds, new imaging techniques are suitable, as 'harmonic imaging'. Often digitized data is postprocessed, as in dynamic angiography, where multiple heart X-ray images are taken prior to contrast media injection, and subtracted from the ones taken after. Computer processing allow the reconstruction of the vascular tree, with a resolution higher than CT or MR angiography.

Medical data is often represented as a volumetric pattern, in the case multiplanar reformatting (MPR) or 3D image reconstruction techniques are applied.

Three dimensional reconstruction includes internal views of blood vessels or bronchi (endoscopic 3-D), or fetal face reconstruction (from standard ultrasound data) in order to detect congenital anomalies. Post processing employs mixed techniques from both medical imaging and computer graphics. This allows detection of 3D structures of interest and new ways of interpretation of generated surfaces. Surgical simulation is a perfect example of an application

much more complex than simple imaging and reconstruction. There, 3D tissues, analogous to that present in surgery, are presented and manipulated with virtual surgical instruments.

At a further level of complexity there is the prediction of surgery outcome, for example, by the simulation of a custom-made implant stress. Such simulation could show the weaker parts, pressure points, and so on, basing on initial data acquisition.

At an ever higher level, image guided surgery is where coordinate system of image data is matched with the patient's one. There, the computers and machinery involved into the treatment process are called 'clinical workstations', and the surgeon could get dynamically cross-sections of the patient, through real time acquiring machinery. These sections could be visualized on a semi-transparent viewing screen, thus giving the surgeon an augmented reality environment to operate in.

The highest stage, nowadays, is the use of surgical robots. Here, images can be used to guide the robot to the point of action, where the surgery will be carried out by the surgeon (passive robot), or the robot itself (active robot). The situations where the supervising surgeon is remotely located are called 'telesurgery' or 'telepresence'. These fields of application are currently far from regular clinical use, and much research is still ongoing.

The need for medical image data interchange led to the development of standards, as DICOM (Digital Imaging and Communication in Medicine), developed by ACR (American College of Radiology).DICOM includes standards for archiving, processing, display, printing, format change, and manipulation of medical digital images and related information, in an integrated environment as an hospital or radiology information system.

3.3.1 A needle-like object penetrating a soft tissue in *UGP*

As an example of a possible use of 3D graphics in medical environment, *UGP* implements a (simple) simulation of an elastic tissue under the pressure of a penetrating needle[4]. The user controls a *needle*, (represented as a gray line), which points towards the center of a tissue portion (on the picture, represented as a mesh with skin-like texturing).

The user can *push* the needle 'inside', thus forcing the tissue to deform. When the needle has reached a certain, 'limit' depth, the tissue 'breaks', leaving the needle inside, while returning to original surface shape.

The represented physical model of simulation is not in any way realistic. It is just a demonstration of how 3D graphics can be used for such kind of animations. Animations can be useful in training medical software, where interactive devices could control virtual surgical tools to perform virtual surgery interventions.

The user would learn to calibrate his manual skills having a visual (and physical, when force feedback is present) response to every action.

Obviously, a specialized software would implement more precise models of interaction and behaviour, beyond the limited scopes of *UGP*.

[4]More informations are in section 9.2.4.

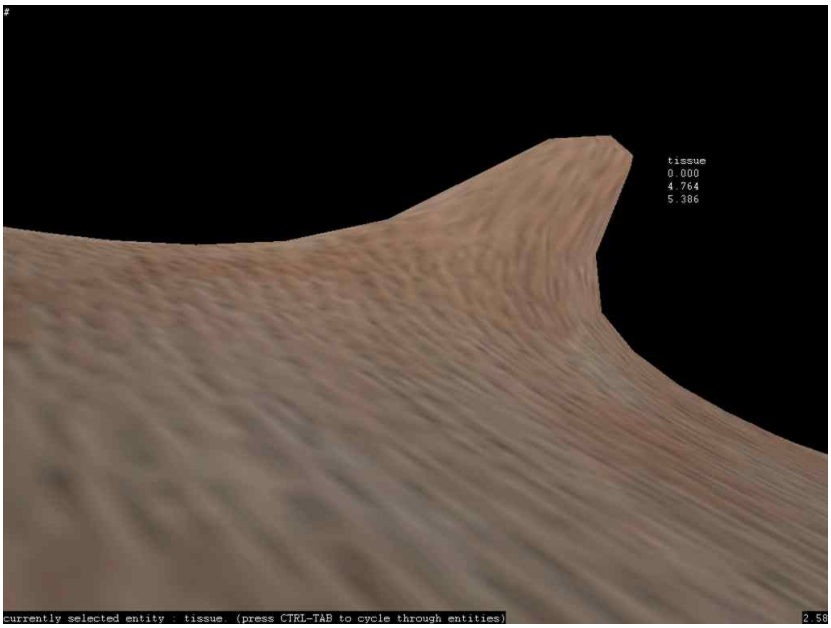

Figure 3.3: *UGP* simulating an elastic, breakable tissue.

Part II

Theoretical foundations

Chapter 4

Two dimensional graphics

4.1 Computer colour and light

When dealing with light and colour, a lot of subjectivity takes place, and in different contexts, different jargons are used. At the beginning of the 20^th century, advantgardist artists started referring in musical terms to colour and visuals [1], and viceversa [MCCM] [2]!

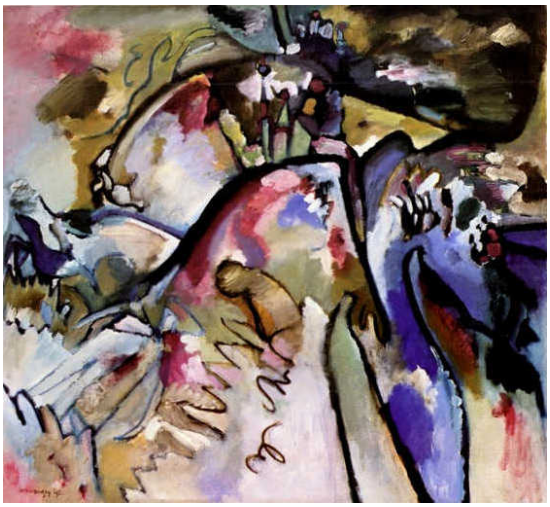

Figure 4.1: Wassily Kandinsky : Improvisation 21a. 1911. Oil on canvas, 96x105 cm. Muenchen, Staedtische Galerie in Lenbach, Germany.

Such synaesthetic[3] attempts are a sound example that in this field terminology is important to not get confused.

Therefore, an appropriated vocabulary is indispensable to distinguish, in light ad colour field, objective aspects from subjective ones. In facts, a correct

[1] Kandinsky's writing Zelenyj zvuk (Green Sound), or paintings called 'Improvisation','Composition','Impression','Concert'.

[2] Schoenberg's Klangenfarbe.

[3] In Greek, syn = union and aisthesis = sensation.

approach refers to the colour as the human perceiving of light.

In the objective analysis, light can be measured through its energy, intensity, wavelength, spectrum, etc.

From the subjective point of view the colour perception possibilities offer much more (this is proved by the fact that the lexicon on the subject of colour is very rich).

Since computer graphics is principally oriented towards human perceptions, special care must be payed to psychovisual aspects of light.

So, experiments conducted on groups of individuals, by various authors, have lead to several *perceptual colour systems*, i.e. attempts to express with a quantitative approach subjective perceptions.

4.1.1 Colour spaces

One of the first perceptual colour models, is the one published by Munsell in 1905 ([WATT], ch.15), describing colours with polar coordinates, with reference to a set of samples (the *Munsell Book of Colour*).

Munsell definition of the coordinates he proposed are:

Hue "It is the quality by which we distinguish one colour family from another, as red from yellow, or green from blue or purple"

Chroma "It is that quality of colour by which we distinguish a strong colour from a weak one; the degree of departure of a colour sensation from that of a white gray; the intensity of a distinctive hue; colour intensity"

(Brightness) Value "It is that quality by which we distinguish a light colour from a dark one"

A further step in this way is the HSV model, by A.R.Smith, in 1978 [WATT]. It bases upon the HSV polar system, or *single hexacone system* in which the colours are bounded in such shape.

In this system, transposing of qualitative characteristics of colour into quantitative is quite easy, since H,S,and V stands for Hue, Saturation and Value.Thus expressions as 'colour X is more yellow than Y' are more natural than in other colour systems.

A major deficiency of this system is the lack of *perceptual independence*, thus, a change in a parameter, say Hue, could influence the change of another parameter. Another aspect, common to other colour systems, is *perceptual non linearity*: equal distances in the space do not correspond always to perceptually equal sensations.

Other colour systems were developed, with diverse target in mind : YIQ for analogue television, for its bandwith efficiency, CMY for editorial practicality, CIE XYZ for general studies about colour perception.

4.1.2 RGB colour space

The human eye retina is gifted with two types of light perceptive cells: cones and rods. The cones are further differentiated, and the differentiated reaction to the spectrum of light among these cells produced a theory claiming the differentiation of three cone types, being each mainly sensitive to red, green, or blue light (in particular, blue light showed being much less stimulating than red and green).

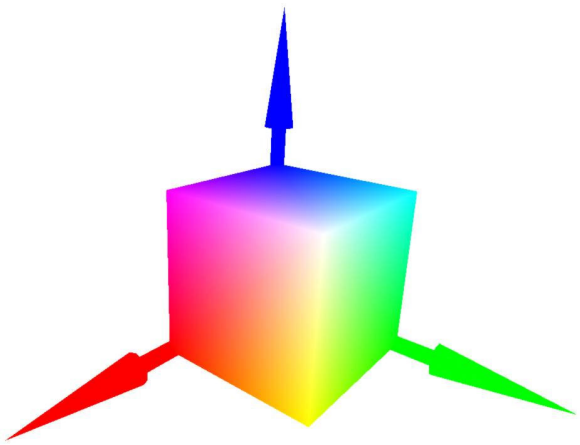

Figure 4.2: A colour construction showing the colour cube in RGB space.

The effects of this theory are intersting : it suggests that any perceivable colour could be obtained through the appropriate combination of red, green and blue light signals. This intuition is not true, but approximations are possible, and this is the reason for existence of the RGB colour model.

In this model [FOLEY], adopted by colour CRT and LCD displays, all the displayable colours are enclosed in a cube defined in the three dimensional space defined by the R, G and B colour vectors. The displayable colours are a subset of the ones perceivable by the human.

Since each display device could have its own wavelength spectrum for the R,G and B components different from another one, the same RGB triples could look differently on different monitors (the phosphors could differ in constructive and function parameters).

Another problem, as non linear perceptivity, and the lack of an intuitive way to imagine the appropriate numerical triple describing a colour make the RGB system a poor colour description system.

Despite this, it is practical enough to be the standard in computer graphics, as most of the computer graphic file formats or imaging devices (cameras, scanners, CCD sensors) make use of it. For the same reasons, the image processing procedures described in the chapter are expressed in terms of the RGB model.

4.2 Transformations and notation

The transformations described in the following sections are a subset ot those implemented in *UGP*.

The notation we will use to indicate a pixel located in column x and row y of the image P, thinking of P as a matrix wide $P.width$ and high $P.height$, is P_{xy}, while we will use P to denote the whole image.

To indicate the red, green, or blue component of the pixel P_{xy}, we will use alternatively $P_{xy}.r \quad or \quad P_{xy}.g \quad or \quad P_{xy}.b$.

When necessary, we will treat a pixel as a vector (in fact it is), writing

expressions as, for example, $P'_{xy} = -P_{xy}$, which negates the red, green, and blue components of P_{xy}.

For simplicity, we make the assumption the R,G, and B, ranges are the *[0,1]* interval.

4.3 Punctual transformations

Punctual transformations are the ones affecting the single pixels without being influenced by the surrounding ones.

The notation we will use for a punctual transformation *punctual* of the pixel $P_{x,y}$ will be :

$punctual(P_{xy}, parameters)$,

where *parameters* could be possible parameters influencing the result of the *punctual* operation.

Nearly every implemented transformation (in the 2D library) can be weighted through one parameter, the *effect* , included in *[0,1]* domain, which makes the resulting pixel as a convex combination of the fully transformed one and the original pixel:

$punctual(P_{xy}, parameters, effect) := effect \cdot punctual(P_{xy}, parameters) + (1 - effect) \cdot P_{xy}$

Since the effect of this weighting is trivial, it will be omitted in the following descriptions, altough it is implemented in the program.

4.3.1 Desaturation

Desaturation consists in subtraction of chroma, and it is implemented by averaging the values of R,G and B channels.

This because the gray colours are positioned along the $(1, 1, 1)$ axis and this transformation aligns the r,g,b values on this axis, keeping the pixel energy (practically, the norm of the colour vector) the same.

$desaturate(P_{xy}).r = desaturate(P_{xy}).g = desaturate(P_{xy}).b := \frac{P_{xy}.r+P_{xy}.g+P_{xy}.b}{3}$ or

$desaturate(P_{xy}) := \underbrace{P_{xy}/P_{xy}}_{=\vec{1}} \cdot \frac{P_{xy}.r+P_{xy}.g+P_{xy}.b}{3}$

4.3.2 Negative

Negation consists in complementing the lightness values of every channel to its maximum. In our notation, the maximum value is 1.

$negative(P_{xy}) := \vec{1} - P_{xy}$

4.3.3 Saturation

Saturation adjustment affects the individual channels distances from the pixel average value.

It is similar to the contrast, but applied to the individual pixels, unrelated.

The effect can be obtained through :

$saturated(P_{xy}) := (saturated(P_{xy}).r, saturated(P_{xy}).g, saturated(P_{xy}).b)$

$saturated(P_{xy}).r := max(min(P_{xy}.r + effect \cdot (P_{xy}.r - desaturate(P_{xy})), 0), 1)$

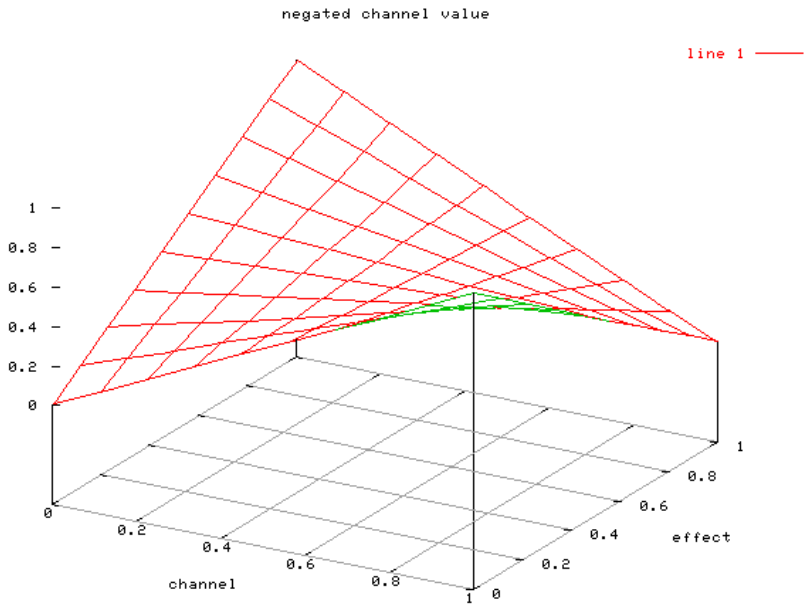

Figure 4.3: Negative.

$$saturated(P_{xy}).g := max(min(P_{xy}.g + effect \cdot (P_{xy}.g - desaturate(P_{xy})), 0), 1)$$
$$saturated(P_{xy}).b := max(min(P_{xy}.b + effect \cdot (P_{xy}.b - desaturate(P_{xy})), 0), 1)$$

4.3.4 Brightness and Vividness

Brightness effect could be reached adding a uniform amount to the R,G,B values, or even multiplying the triple by an increasing factor, say 1.1, such that the maximum component be less than 1. The side effect of the second approach is a small increase of contrast in the image, and could be better referred as a Vividness effect.

$$brightened(P_{xy}, effect) := P_{xy} + effect(1 - min(P_{xy}.r, P_{xy}.g, P_{xy}.b))$$
$$vividized(P_{xy}, effect) := P_{xy} \cdot effect(1/max(P_{xy}.r, P_{xy}.g, P_{xy}.b))$$

4.3.5 Whiteness

Whiteness is the proximity to the white colour in RGB space, and adjusting 'whiteness' could consist in a convex combination of the source pixel with the white colour.

$$white := \vec{1}$$
$$whitened(P_{xy}, effect) := P_{xy} \cdot (1 - effect) + effect \cdot white$$

4.3.6 Blackness

Blackness is the proximity to the black colour in RGB space, and adjusting 'blackness' could consist in a convex combination of the source pixel with the

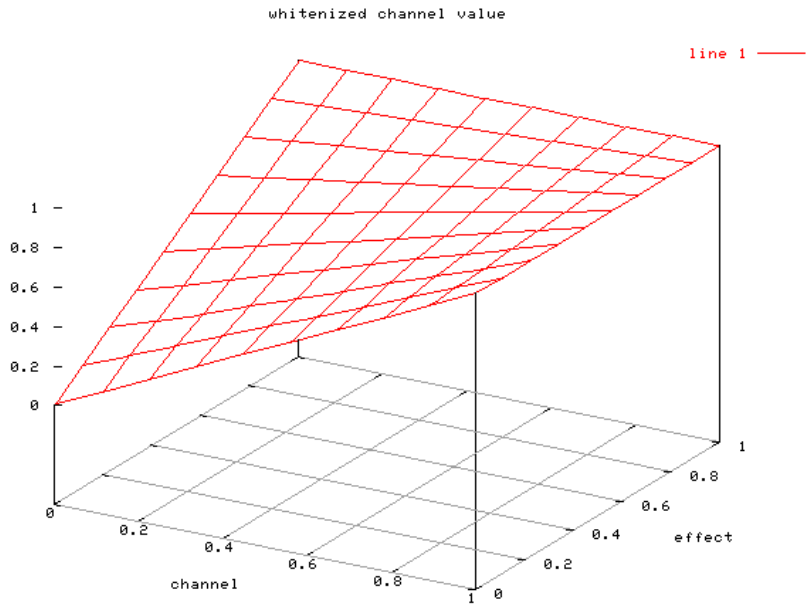

Figure 4.4: Whiteness.

black colour.

$black := \vec{0}$

$blackened(P_{xy}, effect) := P_{xy} \cdot (1 - effect) + effect \cdot black$

4.4 Non Punctual transformations

Non punctual transformations, are the ones in which the final value of a pixel depends on the value of the neighborhood pixels.

They can be sometimes represented as convolutions, or, more precisely, discrete convolutions in discrete, two dimensional space.

In this (computing) context these transformation can be called matrix convolution filters. In the following section convolutions filters are described, and in the following resampling algorithms will be.

4.4.1 Blur and matrix convolutions

Blur operations are the ones which give the original images an out-of-focus fashion.

The blur effect is obtained, for each pixel, through a weighted sum of the surrounding ones, where the weights are decreasing as pixel distance increases.

In fact, each result pixel is a convoluted sum of the surrounding pixels.

For each surrounding pixel an 'influence value' is computed with respect to the target, based on the relative distance from it.

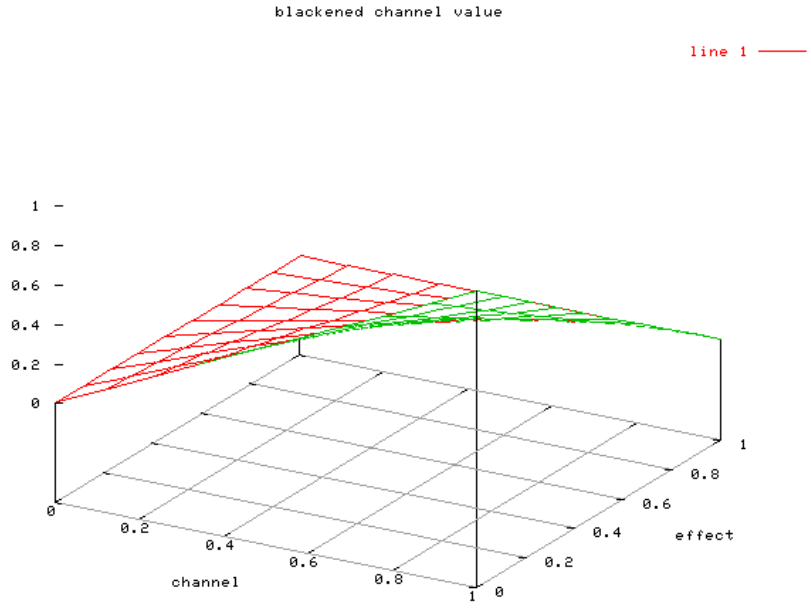

Figure 4.5: Blackness.

Since the source pixels are surrounding the target at fixed distances, the influence values could be easily stored in a (possibly) square matrix of odd order, where the middle entry (central column, central row) should contain the self-influence value for the target pixel itself. This matrix is called the *filter kernel*.

In order to not alter the total energy of the image, the sum of the weights of the pixels in the filter kernel must be equal to 1 (a normalized kernel), or the sum result will have to be divided by the sum of the kernel elements (this is correct if it is assumed that all colour channels carry the same energy, and the colour values are linearly proportional to the energy; in computer graphics it is a weak hypothesis since the energy associated to the values increases exponentially) [FOLEY].

However, since the pixels at the image borders and corners (and nearby, for larger kernels) are convoluted with a smaller number of other pixels (the upper right pixel doesn't have upper and right neighbourhood pixels), some energy loss occurs, and the visible effect is a darkening at the image borders. Alternatively, a smart blur procedure would use different kernels for those pixels, or could work in a 'image wrap' fashion, considering the left upper pixel as the right neighbourhood of the upper right. This last effect is strongly desired in texture creation, where border continuity is a prerequisite.

The filter kernel could be built with the following parameters : the horizontal/vertical (they can differ) distances from the farthest considered pixels, and the function calculating the influence values upon the distances.

A popular blur technique is the gaussian blur filter, whose kernel $K_{gaussian}$

is obtained using the Tartaglia-Pascal triangle, choosing a row R from it, and premultiplying it by its own transponded R^T:

$K_{gaussian} := R^T \times R$

$$
\begin{array}{c}
1 \\
1\ 2\ 1 \\
1\ 3\ 3\ 1 \\
1\ 4\ 6\ 4\ 1 \\
1\ 5\ 10\ 10\ 5\ 1 \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots
\end{array}
$$

Figure 4.6: The Tartaglia-Pascal triangle.

An example of gaussian blur kernel of order 2:

$$
\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 1 \end{pmatrix}
$$

The sum of the coefficients of $K_{gaussian}$ is 16, so the corresponding normalized kernel $K_{gaussian_N}$ is:

$$
\begin{pmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{pmatrix}
$$

The blurred pixel $P_{x,y}$, considering a non normalized kernel K, is computed:
$blur(P_{x,y}, K, radius) :=$

$$
\frac{1}{\sum_{i,j \in [0, 2 \cdot radius]} K_{i,j}} \sum_{i=-radius}^{radius} \sum_{j=-radius}^{radius} P_{x+i,y+j} \cdot K_{i+radius, j+radius}
$$

With a normalized kernel K_N:
$blur(P_{x,y}, K_N, radius) :=$

$$
\sum_{i=-radius}^{radius} \sum_{j=-radius}^{radius} P_{x+i,y+j} \cdot K_{N\, i+radius, j+radius}
$$

In computing, the first form is more used, since often the colour values are expressed as integers, and a lot of floating point multiplications could be less practical to compute than a number or integer multiplications followed by a floating point division.

4.4.2 Contrast

The contrast operation compares each pixel channel's intensity to the average intensity for that channel in the image. If this is lower, the operation lowers it further, through a non-linear function.

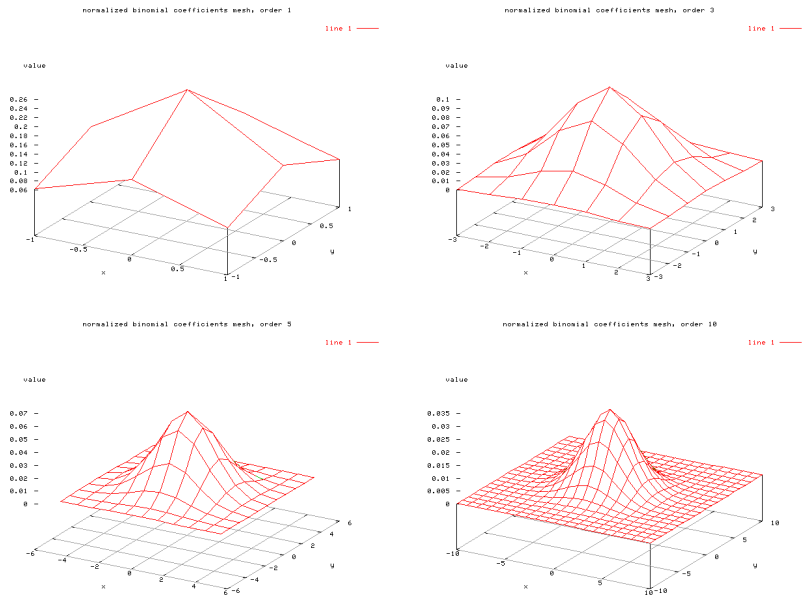

Figure 4.7: Normalized binomial coefficient values for the correspondent kernel.

$$average(P) := \frac{1}{P.width \cdot P.height} \cdot \sum_{x=0}^{P.width-1} \sum_{x=0}^{P.height-1} P_{xy}$$

4.4.3 Despeckle

The Despeckle operation is used to mitigate pixels contrasting too much with the surrounding ones.

The procedure compares the 8 adjacent couples of the nearest 8 pixels surrounding the source pixel, finding the couple with the bigger gap, and memorizing it.

Then, if the gap between the source pixel and any of the 8 neighbourhoods is bigger than the memorized one, the pixel is replaced with the average of the 8 pixels.

This filter is used to eliminate single pixels that could be imperfections in the image.

Enrichment options could introduce a tolerance to stretch/expand the comparison gap, and the possibility to blend the replaced pixel with a percentage of the original one.

In the following notation a helper function $max(pixel, pixel)$, written in declarative style, is used to make the pseudocode more readable. It determines the pixel with the greatest norm (the euclidean norm, treating the pixel as a vector of reals).

$max(pixel \quad P_1, ..., pixel \quad P_N) := P_i \quad s.t. \quad abs(norm(P_i)) \geq abs(norm(P_j)),$
$i, j \in \{1, ..., N\}$

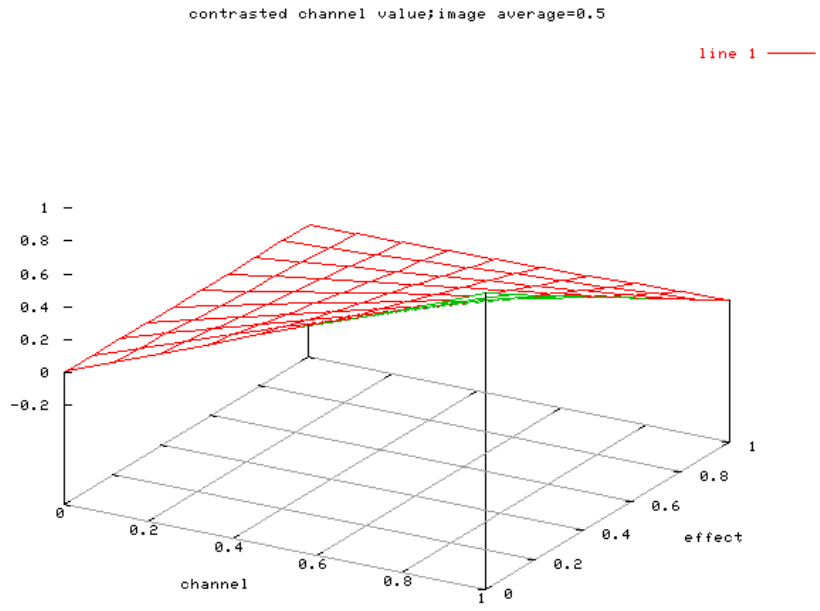

Figure 4.8: Contrast.

$maxSurroundingGap(P,i,j) := max(P_{i+1,j+1} - P_{i+1,j}.P_{i+1,j} - P_{i+1,j-1}.P_{i+1,j-1} - P_{i,j-1}.P_{i,j-1} - P_{i-1,j-1}.P_{i-1,j-1} - P_{i-1,j}.P_{i-1,j} - P_{i-1,j+1}.P_{i-1,j+1} - P_{i,j+1}.P_{i,j+1} - P_{i+1,j+1})$

$maxGap(P,i,j) := max(P_{i+1,j+1} - P_{i,j}.P_{i+1,j} - P_{i,j}.P_{i+1,j-1} - P_{i,j}.P_{i,j-1} - P_{i,j}.P_{i-1,j-1} - P_{i,j}.P_{i-1,j} - P_{i,j}.P_{i-1,j+1} - P_{i,j}.P_{i,j+1} - P_{i,j})$

$if(max(maxGap(P,i,j) > maxSurroundingGap(P,i,j)))$

$then \quad P_{i,j} := \frac{P_{i+1,j+1}+P_{i+1,j}+P_{i+1,j-1}+P_{i,j-1}+P_{i-1,j-1}+P_{i-1,j}+P_{i-1,j+1}+P_{i,j+1}}{8}$

4.4.4 Sharpening

The formula we used to implement gaussian blur, is suitable for other filters too.

In particular, with the proper filter kernel, even the opposite effect can be reached, obtaining a *sharpening filter*.

A sharpening kernel has the peculiarity of null or negative influence values for the pixels surrounding the source. Thus, the only positive entry in the matrix should be the middle one. Some examples follow, for matrices of order 3 (*radius* 1). Note that the kernels are normalized (the sum of the entries is always 1).

$$K^1_{sharpening} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}; K^2_{sharpening} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix};$$

$$K^3_{sharpening} = \begin{pmatrix} -k & -k & -k \\ -k & 8k+1 & -k \\ -k & -k & -k \end{pmatrix}$$

Finally, the matrix convolution formula applied to a sharpening kernel:

$$sharpen(P_{x,y}, K_{sharpening}, radius) :=$$

$$\sum_{i=-radius}^{radius} \sum_{j=-radius}^{radius} P_{x+i,y+j} \cdot K_{sharpening_{i+radius,j+radius}}$$

4.5 Resizing (resampling)

More correctly refferred as resampling, resizing algorithms assign to a given pixelmap P, with size $P.width \times P.height$ a pixelmap P' sized $P'.width \times P'.height$

The following algorithms (upsampling and downsampling) work strictly as they are named.

Thus, the first will only work when both the new width and height are at least bigger than the old ones of a factor equal to 1, and the second if this ratio is at most 1.

To resample an image of size 256x256 to, say, 128x512, a combined procedure would first enlarge the image to 256x512 with the upsample algorithm, with factors 1 for width and 2 for height, and then will stretch the result with factors 0.5 and 1 with the downsampling algorithm.

However, the reverse order of resampling procedures would speed up the procedure, since there were no such a big intermediate image as 256x512 but a slightly smaller 128x256 one.

These algorithms are implemented in the 2d library. The first is a variant of the bilinear interpolation.

4.5.1 Resizing - Upsampling

$\alpha_x = (P.width - 1)/(P'.width - 1)$
$\alpha_y = (P.height - 1)/(P'.height - 1)$
$for(i = 0; i < P'.width; i++)$
    $for(j = 0; j < P'.height; j++)$
        $x = i/\alpha_x$
        $x_0 = \lfloor i/\alpha_x \rfloor$
        $x_1 = \lceil i/\alpha_x \rceil$
        $y = j/\alpha_y$
        $y_0 = \lfloor j/\alpha_y \rfloor$
        $y_1 = \lceil j/\alpha_y \rceil$
        $\beta_x = x - x_0$
        $\beta_y = y - y_0$
        $d_{\nwarrow} = \sqrt{\beta_x^2 + \beta_y^2}$
        $d_{\searrow} = \sqrt{(1 - \beta_x)^2 + (1 - \beta_y)^2}$
        $d_{\nearrow} = \sqrt{(1 - \beta_x)^2 + \beta_y^2}$
        $d_{\swarrow} = \sqrt{\beta_x^2 + (1 - \beta_y)^2}$
        $p_{\nwarrow} = 1 - d_{\nwarrow} / \sqrt{2}$

$$
\begin{aligned}
&p_{\searrow} = 1 - d_{\searrow} / \sqrt{2} \\
&p_{\nearrow} = 1 - d_{\nearrow} / \sqrt{2} \\
&p_{\swarrow} = 1 - d_{\swarrow} / \sqrt{2} \\
&\varepsilon = p_{\nwarrow} + p_{\searrow} + p_{\nearrow} + p_{\swarrow} \\
&\varepsilon_{\nwarrow} = p_{\nwarrow} / \varepsilon \\
&\varepsilon_{\searrow} = p_{\searrow} / \varepsilon \\
&\varepsilon_{\nearrow} = p_{\nearrow} / \varepsilon \\
&\varepsilon_{\swarrow} = p_{\swarrow} / \varepsilon \\
&P'_{ij} = e_{\nwarrow} \cdot P_{x_0 y_0} + e_{\searrow} \cdot P_{x_1 y_0} + e_{\nearrow} \cdot P_{x_0 y_1} + e_{\swarrow} \cdot P_{x_1 y_1}
\end{aligned}
$$

4.5.2 Resizing - Downsampling

$$
\begin{aligned}
&\alpha_x = P.width / P'.width \\
&\alpha_y = P.height / P'.height \\
&\quad for(i = 0; i < P'.width; i++) \\
&\quad for(j = 0; j < P'.height; i++) \\
&\qquad x_0 = i \cdot \alpha_x \\
&\qquad x_1 = (i+1) \cdot \alpha_x \\
&\qquad y_0 = j \cdot \alpha_y \\
&\qquad y_1 = (j+1) \cdot \alpha_y \\
&\qquad P_{ij} = null \\
&\qquad for(k = \lceil x_0 \rceil; k < \lfloor x_0 \rfloor; k++) \\
&\qquad\qquad for(l = \lceil y_0 \rceil; l < \lfloor y_0 \rfloor; l++) \\
&\qquad\qquad\qquad P'_{ij} = P_{kl} \\
&\qquad for(k = \lceil x_0 \rceil; k < \lfloor x_0 \rfloor; k++) \\
&\qquad\qquad P'_{ij} += P_{k\lfloor y_0 \rfloor} \cdot (\lceil y_0 \rceil - y_0) \\
&\qquad\qquad P'_{ij} += P_{k\lfloor y_1 \rfloor} \cdot (y_1 - \lfloor y_1 \rfloor) \\
&\qquad for(l = \lceil y_0 \rceil; l < \lfloor y_0 \rfloor; l++) \\
&\qquad\qquad P'_{ij} += P_{\lfloor x_0 \rfloor l} \cdot (\lceil x_0 \rceil - x_0) \\
&\qquad\qquad P'_{ij} += P_{\lfloor x_1 \rfloor l} \cdot (x_1 - \lfloor x_1 \rfloor) \\
&\qquad \beta_0 = (\lfloor x_1 \rfloor - \lceil x_0 \rceil) \cdot (\lceil y_0 \rceil - y_0) \\
&\qquad \beta_2 = (\lfloor x_1 \rfloor - \lceil x_0 \rceil) \cdot (y_1 - \lfloor y_1 \rfloor) \\
&\qquad \beta_3 = (\lceil x_0 \rceil - x_0) \cdot (\lfloor y_1 \rfloor - \lceil y_0 \rceil) \\
&\qquad \beta_1 = (x_1 - \lfloor x_1 \rfloor) \cdot (\lfloor y_1 \rfloor - \lceil y_0 \rceil) \\
&\qquad \varepsilon_0 = (\lceil x_0 \rceil - x_0) \cdot (\lceil y_0 \rceil - y_0) \\
&\qquad \varepsilon_2 = (\lceil x_0 \rceil - x_0) \cdot (y_1 - \lfloor y_1 \rfloor) \\
&\qquad \varepsilon_1 = (x_1 - \lceil x_1 \rceil) \cdot (\lceil y_0 \rceil - y_0) \\
&\qquad \varepsilon_3 = (x_1 - \lceil x_1 \rceil) \cdot (y_1 - \lfloor y_1 \rfloor) \\
&\qquad P'_{ij} += P_{\lfloor x_0 \rfloor \lfloor y_0 \rfloor} \cdot \varepsilon_0 \\
&\qquad P'_{ij} += P_{\lfloor x_0 \rfloor \lfloor y_1 \rfloor} \cdot \varepsilon_2 \\
&\qquad P'_{ij} += P_{\lfloor x_1 \rfloor \lfloor y_0 \rfloor} \cdot \varepsilon_1 \\
&\qquad P'_{ij} += P_{\lfloor x_1 \rfloor \lfloor y_1 \rfloor} \cdot \varepsilon_3 \\
&\qquad \gamma = (\lfloor x_1 \rfloor - \lceil x_0 \rceil) \cdot (\lfloor y_1 \rfloor - \lceil y_0 \rceil) \\
&\qquad P'_{ij} /= (\beta_0 + \beta_1 + \beta_2 + \beta_3 + \varepsilon_0 + \varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \gamma)
\end{aligned}
$$

Chapter 5

Three dimensional graphics

This section introduces some fundamental concepts used when dealing with computer graphics.

5.1 Points, planes, and straight lines

Localization in Euclidean space is obtained through a three axis coordinate system, associating a triple of real numbers (x,y,z) to each point (or *vertex*).

Two non-coincident points locate univocally a straight line, and the straight line portion between them is a segment. The normalized difference vector between any two points laying on a straight line gives us a vector representing the inclination of the straight line.

This oriented vector is also referred as gradient.

Gradient information is equivalent even if its (l,m,n) values (see below) are multiplied by a common factor, as long as proportionality of its components is the same. Thus, we can represent a straight line with a vector $D := (l, m, n)$ as its direction, plus a base starting point $P := (x_0, y_0, z_0)$ lying on it. Any point (x, y, z) laying on the given straight line respects the equations :

$$\begin{cases} x = x_0 + l \cdot t \\ y = y_0 + m \cdot t \\ z = z_0 + n \cdot t \end{cases} \tag{5.1}$$

In particular, the parameters of a straight line can be extrapolated from two points P_1, P_2 and treating $P_2 - P_1$ as D and P_1 as P.

A straight line in space can also be seen as the intersection of two planes.

The notation for a plane is made up of its normal (orthogonal) vector $N := (a, b, c)$, and a scalar parameter d, or the equivalent quadruple.

The plane (a, b, c, d) is the set of points (x, y, z) for which is true $a \cdot x + b \cdot y + c \cdot z + d = 0$.

Since N is normal to the plane, the plane closest point to the origin $(\vec{0})$ is $P_{nearest} := (x_n, y_n, z_n)$, and will be oriented towards the N direction, and thus, for a certain α, $P_{nearest} = \alpha \cdot N = (\alpha a, \alpha b, \alpha c)$.

Follows :

$a \cdot \alpha a + b \cdot \alpha b + c \cdot \alpha c + d = 0 \Rightarrow$

$\alpha a^2 + \alpha b^2 + \alpha c^2 + d = 0 \Rightarrow$

$\alpha(a^2+b^2+c^2)+d=0 \Rightarrow \alpha = \frac{-d}{(a^2+b^2+c^2)} \Rightarrow P_{nearest} = \alpha N = \frac{-d}{(a^2+b^2+c^2)}N = \frac{-d}{\|N\|^2}N$

If considering the plane nearest point to another generic point in space, P, the reasoning is analogous, and only a displacement of a projected vector is necessary to be added:

$P_{nearest}(P) = \frac{-d}{\|N\|^2}N + (P + \Pi_{<N>}(P)) = \frac{-d}{\|N\|^2}N + (P + \frac{N \cdot P}{\|N\|^2}N) = \frac{P \cdot N - d}{\|N\|^2}N + P$

5.1.1 Containment of points in polygons

A plane is univocally determined by three non-aligned points in space, in a certain order.

A triple of points (P_1, P_2, P_3) define a plane, and the space among them is a triangle.

Numerically, the triangle is a subset of the span of the three points, with the constraint that the sum of the point coefficients must be equal to 1 (and any coefficient not less than 0). More precisely, the triangle is the convex hull of the three points.

This property gives us a way to check if a point belongs to a triangle.

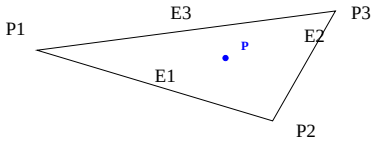

Figure 5.1: A point belonging to a triangle.

Solving the matrix associated to 3 points coordinates as columns, the coefficient column as the unknown, and the result column as the point to check, we check whether the sum of coefficients in unknown column equals 1.

$$\begin{pmatrix} P_1.x & P_2.x & P_3.x \\ P_1.y & P_2.y & P_3.y \\ P_1.z & P_2.z & P_3.z \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix} = \begin{pmatrix} P.x \\ P.y \\ P.z \end{pmatrix}$$

P is in the triangle delimited by $P1, P2, P3$ iff $(\alpha_1+\alpha_2+\alpha_3 = 1; \quad \alpha_1, \alpha_2, \alpha_3 \geq 0)$

In practice, a small tolerance can be accepted when doing this kind of checks when computing with floating point numbers, due to unavoidable numerical errors.

A point laying on a generic n-polygon has also the property $\sum_{i=1}^{n} \alpha_i = 1$, but for practical uses this property is less popular for checks, because the input points could not lay on the same plane. Besides this problem, the following system would have more than one solution, so other techniques should be used to determine a suitable one:

$$\begin{pmatrix} P_1.x & P_2.x & P_3.x & ... & P_n.x \\ P_1.y & P_2.y & P_3.y & ... & P_n.y \\ P_1.z & P_2.z & P_3.z & ... & P_n.z \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ ... \\ \alpha_n \end{pmatrix} = \begin{pmatrix} P.x \\ P.y \\ P.z \end{pmatrix}, (\alpha_i \geq 0, i \in 1, .., n)$$

For practical reasons, the polygons are usually split in triangles and then one proceeds as with the first method, iterating for every polygon.

An alternative containment check computes the sum of the angles between the lines from the point to each vertex. If the sum equals 360 deg, the point is contained in the polygon.

5.1.2 Ray - polygon intersection

The ray-polygon intersection is calculated in three steps [WATT] :

1. obtaining the equation for a plane containing the polygon
2. checking for the intersection of this plane with the ray
3. checking that this intersection is inside the polygon

The parametrical form of a ray (an oriented segment) from point R_1 to R_2, is:

$R = R_1 + (R_2 - R_1)t$ is equivalent to

$R.x = R_1.x + (R_2.x - R_1.x)t = R_1.x + it$

$R.x = R_1.y + (R_2.y - R_1.y)t = R_1.y + jt$

$R.x = R_1.z + (R_2.z - R_1.z)t = R_1.z + kt$

, where $0 \leq t \leq 1$

The point R will lay on both the ray and the polygon plane (A, B, C, D) if the preceding equation and the plane equations are satisfied :

$A \cdot R.x + B \cdot R.y + C \cdot R.z + D = 0 \Rightarrow$

$A \cdot (R_1.x + it) + B \cdot (R_1.y + jt) + C \cdot (R_1.z + kt) + D = 0 \Rightarrow$

$A \cdot R_1.x + B \cdot R_1.y + C \cdot R_1.z + D + t(A \cdot i + B \cdot j + C \cdot k) = 0 \Rightarrow$

$t = \frac{-(A \cdot R_1.x + B \cdot R_1.y + C \cdot R_1.z + D)}{(A \cdot i + B \cdot j + C \cdot k)}$,with $(0 \leq t \leq 1)$

If the t is negative, the plane is behind the ray.

If it is greater than 1, the plane is in front the ray, but over R_2.

If it is in the $[0, 1]$ interval, the plane intersects with the ray.

The ray-polygon intersection exists if the computed intersection point lays in the polygon.

This condition is computed with any polygon containment check.

5.1.3 Back and front polygons

Given a triangle bounded in P_1, P_2, P_3, we can find the parameters of its plane (a, b, c, d), by the fact that $N = (a, b, c)$ is the vector normal to the plane, and it is obtainable by $N = V_1 \times V_2$, with V_1, V_2 being two vectors parallel to the plane. These can be chosen as $V_1 := P_2 - P_1, V_2 := P_3 - P_1$, (or even the negated ones : $V_1 := P_1 - P_2, V_2 := P_1 - P_3$)

The parameter d is easily calculated, as $d = -(a \cdot P_x + b \cdot P_y + c \cdot P_z)$, with P being any point on the plane, such as P_1, P_2,or P_3.

Note: if we swap V_1 and V_2 (or the swapped and negated), the resulting normal N would be the negated of the previous one. Also the d parameter would be negated. This leads to the conclusion that with a triple of points we can locate two planes, depending on the vectors to use for the normal.

This fact is used in computer graphics for introducing the concept of *face culling*.

In a rendering system, when dealing with triangles, it is possible to draw them on screen only if their orientation to the viewer make them 'visible'. We can think of the polygon as 'visible' when its normal is oriented toward the viewer.

So, when the angle of the view vector V with the polygon normal vector N is in the $[\frac{\pi}{2}, \frac{3\pi}{2}]$ interval. This occurs when the cosine of this angle is negative.

Since $cos(V, N) = \frac{V \cdot N}{\|V\|\|N\|}$, and the denominator is always positive, the desired condition for a *visible* face is $V \cdot N < 0$.

This concepts are usually employed for hidden surface removal (face culling), lightning calculation tasks, or collision tests.

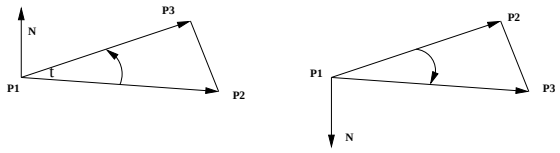

Figure 5.2: Triangle orientation.

5.2 Meshes

The most popular way in computer graphics to represent three dimensional objects employs a net, or *mesh* of planar polygonal facets.

In this application, the only polygons used are triangles. This choice was made in regards to computation ease and rendering facilities offered by a triangle representation. Therefore, the mesh term will always refer to a triangle mesh. We associate to a mesh : a set of E edge elements, a set of V vertex elements, and a set with T triangle elements. An additional property of the meshes used in this application is that they are *hulls*.

5.2.1 Hull properties

Imagine to have some square carton sheets. To join them to form a closed box you have to order them such that every sheet border touches another just one time. If even only one border remains unjoined, there's an evident 'hole' in the structure.

Back to our mesh representation, if there exists at least one edge not shared by a pair of triangles, the mesh will appear 'open', or, otherwise, not as a *hull*.

Another constraint, occurring when triangles are oriented (we can choose to not orient them, making them visible from both sides), is that two adjacent triangles, when calculating their normals, must consider the shared edge orientation in opposite orientations. If the edge is considered oriented in the same way by both triangles, the normals will have an angle in $[0, \frac{\pi}{2}]$ when the angle between the planes is in the $[\frac{\pi}{2}, \pi]$ interval and viceversa (the normals will have an angle in $[\frac{\pi}{2}, \pi]$ when the angle between the planes will be in the $[0, \frac{\pi}{2}]$ interval). The visual effect occurring in the rendered image, with back face elimination enabled, will be 'holes' where triangles should be, just as triangles were missing.

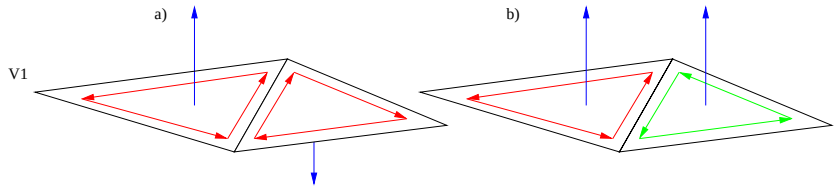

Figure 5.3: Incorrectly (a), and correctly (b) ordered edges.

Thus, the following properties are valid for every hull mesh, but could not be valid for any generic mesh.

An edge is localized by a couple of vertices.

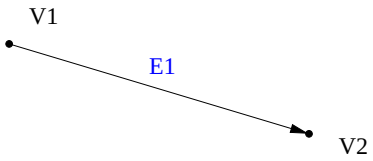

Figure 5.4: An edge.

If three non-coincident edges have three points in common, two by two, then the edges define a triangle.

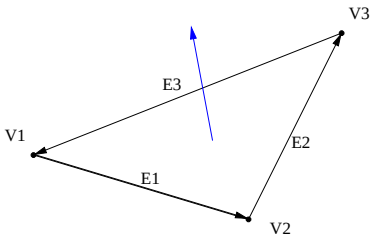

Figure 5.5: A triangle.

In a mesh representation with the hull constraint, each edge should appear exactly in 2 triangles (or we can say that the two triangles have an edge in common).

And each triangle shares exactly 3 edges (and vertices) with other ones.

5.2.2 Some properties of triangle meshes

Given a mesh representing an object (it must be a closed mesh), with V vertices, E edges connecting them, we have always

$E = 3 * (V - 4) + 6$ and

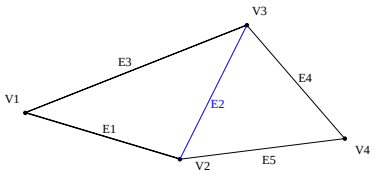

Figure 5.6: A triangle enclosed by three edges.

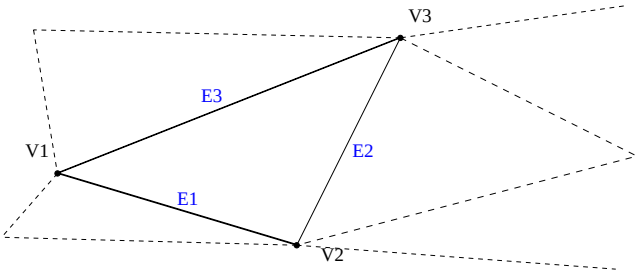

Figure 5.7: Edges sharing in adjacent triangles.

$T = 2/3 * E$, and thus
$T = 2V - 4$

The second property derives from the fact that each edge appears only 1 time in 2 triangles.

The first is a bit trickier, but can be proved by induction, beginning from the smallest polyhedron, the tetrahedron (for example, a pyramid with triangular basis), with $E = 6, V = 4, T = 4$, as shown in the following figure.

By adding to it an edge E_8 from vertex V_1 to edge E_6 (thus halving a face), we must take care to split E_6 edge in two, and split the other triangle touching E_6, and we get new edges E_7, E_9, and two new triangles. Now it is E=9, V=5, T=6.

We increased the vertex count by 1, the triangle count by 2 and the edge count by 3. If we continue, we discover that every new vertex brings 3 edges and 2 triangles to the mesh.

The important thing to keep in mind when proving this is to verify whether each vertex lays on the end of an edge. If not, we split in two the edge and add another one, symmetrically to the split edge. Obviously, when adding another edge, we place its end on a vertex (otherwise we would continue endlessly).

So far, we have found that the triangle count increased by 2 and the edge count by 3 (1 for the new edge, one from the split edge, and one from the additional edge)! The third fact is derived from the first two.

5.2.3 Convexity

A mesh is convex if no triangle's front face normal, treated as a ray, hits any triangle face.

In other words, every mesh triangle plane partitions the space in two subregions: one containing all vertices from the mesh, another one with no mesh vertices.

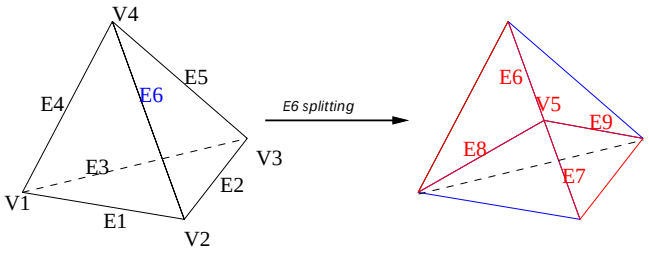

Figure 5.8: A tetrahedron with split faces.

If any of the triangle planes violates this condition, the mesh is not convex.

A consequent property of convexity is that every triangle plane normal, if inverted, has the center of the mesh in the positive subspace ('front' region).

Moreover, if the mesh is convex, there exists no such triangle couple with their external angle negative. Visually, all of the normal vectors pointing externally from the mesh are divergent.

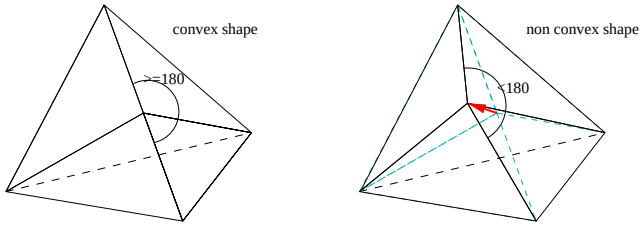

Figure 5.9: A convex and a non convex mesh.

5.2.4 The implemented data structure

The data structure implemented for mesh representation is hierarchical and a little redundant.

It is based on an array (an ordered set) of edges E, one of vertices, V, and one of triangles, T. The vertex data structure holds coordinates, and possibly colour information. The vertices are stored in the V array.

Each edge data structure holds the indexes for the correspondent vertices in the V array. The edge is oriented from the first vertex to the second one.

Finally, each triangle structure has indexes to both triangles and edges in their corresponding arrays (3 vertices and 3 edges).The indexes (or logical pointers) are ordered.

Since each edge in E array is oriented and shared by two triangles (for hull consistency), one triangle will consider the edge as it is declared, and the other will consider it as inverted.

In the figure 5.10 is shown a partial structure for a tetrahedron.

The 'inverted' or 'as it is' condition is represented in the data structure as a boolean value. In the figure the use of the edges for the triangle is marked as '1'

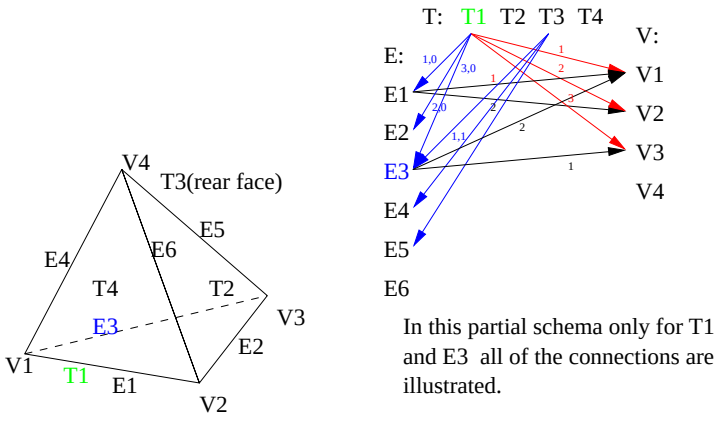

Figure 5.10: An example of mesh representation.

for inverted and '0' for 'as it is'. The numbers indicate the correct order of the vertices/edges in the triangle.

It is shown that the $E3$ edge is shared by $T1$ and $T3$, having it labeled $T1$ as 'as it is' and $T3$ as 'inverted' (B). This because $E3$ originally refers to the ordered couple ($V1$, $V2$), and $T3$ needs to consider it as ($V2$, $V1$).

A pseudocode implementation of the structures;

structure Vertex(Real x,Real y, Real z,Colour c)

*structure Edge(Vertex *v1,Vertex *v2)*

*structure Triangle(Vertex *v1,Vertex *v2,Vertex *v3,Edge *e1,Edge *e2,Edge *e3,Bool i1,Bool i2,Bool i3)*

structure Mesh(Vertex[] V,Edge[] E,Triangle[] T)

The main advantage of this mesh structure is flexibility : more vertices can be selected for translation/rotation/other transformations via the triangles they belong to.

If the triangle data would be duplicated in each triangle, such operations would necessitate to be replicated for each copy, and in case of an error, the chance of non-congruence between triangle edges could occur.

The possibility of manipulation makes this structure good for hierarchical body animation, where a certain organization of vertices is essential.

When it comes to rendering, the most popular graphics hardware is optimized to display triangle point triples, and these are directly obtained from the T array.

For wireframe rendering, it is possible to refer directly to the E array, avoiding the duplication that would occur when drawing edges indirectly, via T. Analogously, for dot rendering, a direct use of the V array avoids the triplication of vertices that would occur when using T array triangles vertices.

Part III

Software architecture

Chapter 6

Introduction

This part will explain the architecture of *UGP*, after some words on the programming style and related issues.

6.1 Programming style

Having done the work in C++, I pursued the style of encapsulating the whole code in classes, thus exploiting object orientation benefits. Although the class determination proceeded quite naturally, the process of correct encapsulation of data structures gave several problems, due mostly to two reasons :

1. the flexibility and performance needs
2. the constraints given by the graphic APIs commands

Since the software got to have real time rendering capabilities, a lot of effort was given to a proper optimization of the code, and good programming habits, like access methods use, sometimes were not possible. This solution would cause the code execution to slow down, if incorrectly optimized.

So, I opted for trade-offs between performance and proper C++ programming style. Practically, this resulted in a certain amount of *friend* declarations of classes and methods.

Moreover, the software requirements for flexibility (the main purpose of the *UGP* software is experimentation with real time rendering), gave further blur to the class responsabilities boundaries.

This lack of internal coherence is the result of the workarounds made to maximize the program flow efficiency, but since the software was built with extension in mind (mostly based on inheritance), the extension interfaces are well defined and consolidated (see section 9.1.1).

The code makes direct use, aside of standard C++ libraries, of the following cross platform libraries:

SDL (Simple DirectMedia Layer),v.2.7 [SDL] and **OpenGL** [WRIGHT],[SGI] (1.1 and beyond).

This choice allows portability in Windows and Linux environments (where the software was tested) and, accordingly to SDL documentation, in *BSD, MacOS, and BeOS too.

6.2 *UGP* components

The main classes of *UGP* are described in the following sections, and they are summarized in the following picture.

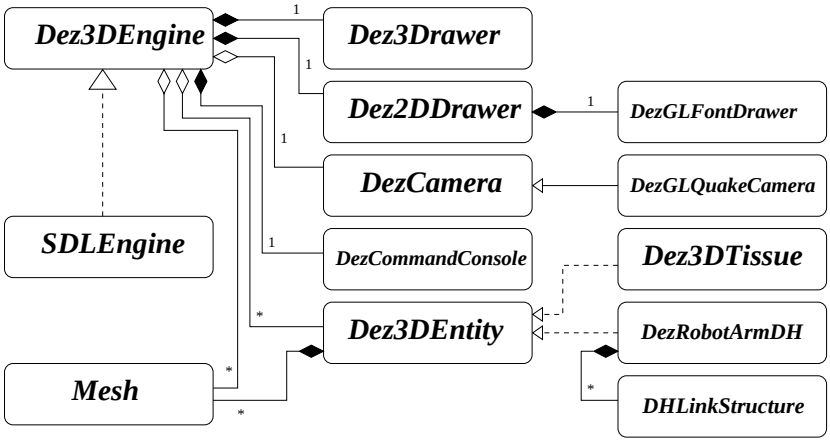

Figure 6.1: The main component classes of *UGP*.

The *Dez3DEngine* class (in fact, a *singleton* class) has the role of a container for all the objects, and encapsulates the program control flow.

Certain tasks, like rendering, viewing system management, input device management, are implemented in separate classes, so an adequate separation of responsabilities defines each class purpose.

The mutual interaction of these (core) classes is strict, and their code sometimes contain member data cross references (mostly in time critical functions, being them the program bottleneck) that makes maintenance and debugging a little harder.

However, when critical performance attention was not necessary (in methods which use is relatively rare), focus was given on a correct organization and code clarity.

6.2.1 *UGP* components by dissection

Dez3DEngine is an aggregate of four tightly related classes: *Dez3DDrawer*,-*Dez2DDrawer*, *DezCamera*, and *Dez3DCommandConsole*.

The

The four classes correspond to four objects aggregated into the *engine* object (there is a 1 to 1 relation between them and the engine class), and their lifetime is the same as its.

However, the *DezCamera* object has a little weaker bound to the engine, since it is replaceable at any time with any class in its hierarchy [1].

The forementioned classes are the service classes which make the engine work.

[1] See section 8.1.5.

The purpose of *UGP*, making use of this code, is to experimentate with graphics, and this goal is reachable when the behaviour of the program is customizable enough with the minimal effort.

With this goal in mind, all the classes without managing/control tasks are the ones managed by the software. The *UGP* program, then, can be thought as the environment managing these classes objects, called *entities*.

All of them are in the inheritance hierarchy starting with the *Dez3DEntity* class. In the previous picture were shown two entity classes, Dez3DTissue and DezRobotArmDH, but more ones exist, and are described in ch. 9.2.

Their interface is made in such a way, that the minimal effort is needed for programming new entities for experimental purposes.

As described later, the *Dez3DEngine* class, being a *virtual* class, has deficiencies (input handling and window API managing missing) covered by a specialized class, *SDLEngine*, described in 7.2.

6.2.2 Minor components and accessory classes

The preceding figure didn't show the data structures keeping the graphics data (meshes, geometry, images).

These were developed separately, in statically linked libraries, called *Dez2D* and *Dez3D*, with, respectively, functionalities for two and three dimensional graphics.

Dez2D library supports raster image loading, in *bmp 24 bit* format only, because more image formats could easily be supported with other library use.

The class holding the image data and image processing data is *DezImage*, and relies on the pixel abstraction implemented in *Dez2DPoint*. This is the specialization (for the *double* C++ type) of the *Dez2DPointT template* class, which holds pixel data and implements basic pixel processing (punctual transformations).

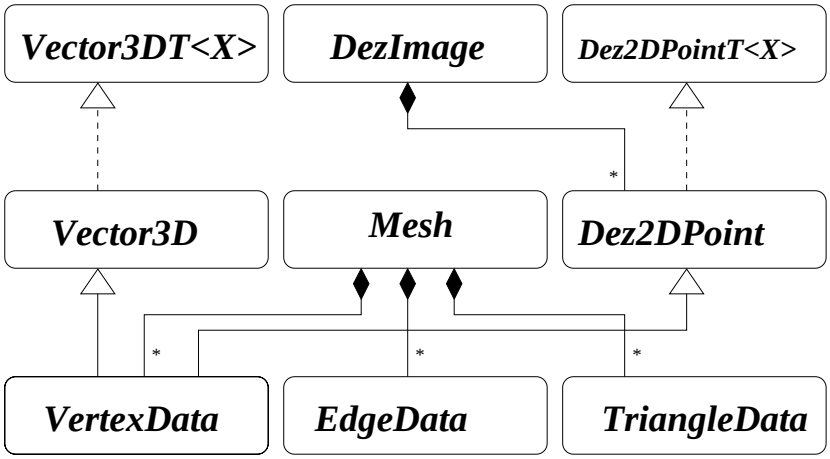

Figure 6.2: Accessory classes.

The three dimensional graphics data structures make use of the two dimensional ones, when dealing with colour information (a *Dez2DPoint* object is

member of *VertexData*), or image information (*DezImage* is used in the drawing classes).

Vector3D<X> is the template class for a three dimensional vector, and instantiated for the *double* type makes up the *Vector3D*.

The direct expansion is *VertexData*, which adds the colour information, otherwise missing, to the *Vector3D* type.

Put together, a lot of *VertexData* objects, related in the appropriate way, can represent a triangle mesh, but not without the appropriate structures representing the fundamental edge and triangle concept (*EdgeData* and *TriangleData*). These two classes, with the use of pointers, realize the concept of *mesh* as described in section 5.2.4.

So, the *Mesh* objects are composed of *TriangleData*, *EdgeData* and *TriangleData* objects, and are the structures used by the engine for the geometry.

To produce the meshes correspondent to primitive shapes, there is the *stateless MeshCreator* class, which contains methods to create various *Mesh* objects, described by a command string as input. Its description is located in chapter 10.1.

6.2.3 The components and libraries

Direct use of the software libraries with graphics and input capabilities is a necessity of this kind of programs. However, an appropriate way of arranging the code gives the possibility to a future porting to other similar libraries.

With this goal in mind, the *SDLEngine* was made the only class which uses the SDL calls directly.

This choice was quite easy to implement, since SDL functions relate to input, which by its nature is directed to the engine object, and graphics (**OpenGL**) initialization and window management, functions executed rarely in the program.

With **OpenGL** this was possible in a more limited way.

The classes using **OpenGL** are the ones with drawing capabilities : *Dez3DDrawer*, *Dez2DDrawer*, *DezGLFontDrawer* (aggregated to *Dez2DDrawer*), *DezCamera* (for reasons explained in 8.1.5), and *Dez3DEntity*.

The drawing classes make use of **OpenGL** for evident reasons, but classes such as *Dez3DEntity* could also do not.

The reason entities can make direct use of **OpenGL** is that they could have some special data objects not drawable by the engine, or could implement some specialized (or optimized) drawing code.

Future implementations of the program will probably discourage the use of **OpenGL** in derived *Dez3DEntity* code.

Actually, *Dez3DEngine* makes use too of **OpenGL**, in the *renderScene()* method, but it is not a problem since the method is virtual, and in future, the **OpenGL** code will migrate to *SDLEngine* or *Dez3DDrawer*.

6.3 Code documentation

Detailed code documentation is maintained in html,pdf,ps formats in the code subdirectories.

The documentation is generated with automatic tools when the code is modified, so the most updated description available is there.

Chapter 7

Engine and control

7.1 The base engine class and inheritance

Here the notion of ”engine” stands for indicating the organized and structured set of data, functions/algorithms used to control, manage the world dynamics, and finally visualize, 3d objects on a 2d screen.

The engine abstraction was implemented in the virtual *Dez3DEngine* class, referred for simplicity as engine.

The basic engine class, *Dez3DEngine*, doesn't handle input for simplicity/portability reasons, but offers methods to be overridden and implemented making use of some chosen API.

Dez3DEngine was programmed as a base class for derived engines, specialized in other tasks (games, small simulators, viewers, small editors, etc).

The basic facilities (object drawing and object managing) are kept general the most possible, in the hope to be portable enough to be used as a skeleton for future, heterogeneous applications.

In *UGP* case, the employed extended class of *Dez3DEngine* implementing input is called SDLEngine, and uses SDL C functions to get the input and initialize **OpenGL**.

Since it is task of the derived engine class to initialize the video subsystem, the basic class doesn't instantiate a *Dez3DDrawer* object, letting the former to create it, when it is possible (when eventual system check indicates the presence of an adequate base software platform).

7.2 The derived class SDLengine

As said, *Dez3DEngine* is a virtual class, because generality needs led to the choice to not implement there the input, but to define all the interface likely to appear in an interactive graphics program of this kind.

The *SDLEngine* class is the specialized version of *Dez3DEngine* class used in *UGP*, and carries the tasks of input handling and **OpenGL** initialization.

7.2.1 OpenGL initialization

At construction time, an *SDLEngine* object has to initialize some base class members necessary for drawing with **OpenGL**.

This operation has sense only if the proper *rendering context* is set up. The *rendering context* is a software abstraction of the data structures between the program and the window system (and the underlying graphic devices).

To create one a rendering context for **OpenGL**, the code should call routines of the window system, and register the application for the video resources (and this is not always possible : imagine a fullscreen application already running and using some exclusive buffers of the graphics device; another application could not have the same resources available).

The knowledge of the window systems was beyond the scope of this work, (aside from being complicated), and the opted solution was the use of the cross platform SDL library.

The *SDLEngine* constructor attempts the initialization of the **OpenGL** subsystem, asking the window system (via the *SDL_VideoModeOK(...)* function) for the availability of the video mode selected f the program (the video mode is defined by horizontal and vertical resolution, and bit depth of the pixels).

If the video mode selected by the user (it can b selected in the initialization file) is not supported, the program will not try to use, but to select another working video mode. If any video mode is not available, the program will fail to create a working **OpenGL** context, and will have to terminate.

If the desired video mode was appropriate, the context is created with a call to *SDL_SetVideoMode(...)*.

The video setting up can be completed, with a series of calls to set the **OpenGL** state concerning the view system and drawing options.

7.2.2 Input handling

Input registration happens with a *SDL_WM_GrabInput(...)* call, and allows the program to read input with SDL facilities.

The input reading is a task performed frame by frame, from within the virtual method *input()* .

Since the *UGP* program has two operating modes (*normal mode* and *entity mode*), there are two functions for grabbing input, one for each mode. To select the method to call to grab input, the program sets an appropriate function pointer to one of the two methods, and calls it until state is changed.

The two methods are *readInput()* (input is interpreted by the engine) and *lightInput()* (input is interpreted by the currently selected entity)

On the inside, the methods are similar, and consist in a poll cycle to the window managing systems, asking for *events*. Every captured event (in form of a *SDL_Event* structure) is recognized and executed, if associated to an action.

The event queue waiting for the program is emptied, having executed all the commands associated to the events, and the function terminates successfully.

The events for which the SDL library is sensitive are key presses, key releases and mouse moves, presses and releases.

The *lightInput()* method is similar, but instead of having the binding between all user events and actions, it has only a part of these. The sensed events are

recognized and fill a little data structure, passed to the current entity as an *input vector*, which will drive the entity behaviour.

Aside of these input grabbing functions, other pure virtual functions exist in the base class, and force the implementation of an appropriate mechanism to switch the modes when needed (for example, deleting an entity will make impossible to having it currently selected, and the input will have to be switched again to the engine).

These functions are the virtual ones *inputToEngine()*, *inputToOthers()*, *inputSwapped()*.

7.3 The command console and control

Given the experimental/didactic nature of the developed software, there was a great need of an expressive way of text based user input to the engine.

Similarly to many videogames, or CAD systems, there is a console command system enabling the user to communicate typed commands to the software.

When the engine is in command console mode, the typed keyboard input is stored in a text buffer in a *Dez3DCommandConsole* object.

When the user presses the submitting key (the Return (enter) key), the text is split into words (tokenized), and sent for recognizing to the engine.

In the engine, the tokenized command is compared with various command patterns, to match the proper one. Once a pattern is found, eventual numerical parameters are evaluated, and the command executed. The commands can be directed to specific objects, through specifying the proper internal identifier as the first command word.

For example, if the command relates to rendering options, it will be sent by the engine to the object responsible for the rendering, and therein interpreted and executed.

In this manner, a common interface among different class objects allows the communication of the user with the object responsible for a particular task, eventually executed using polymorphism (a hierarchy of classes could recognize the same command with the base class interpreter and execute them differently).

For example, commands not handled by a derived engine class (in the redefined command interpretation method) can be sent to the base class method (with the proper C++ casting).

The engine method interpreting console commands are *Dez3DEngine::executeLine(char* commandLine)* for unprocessed commands and *Dez3DEngine::command(int argc, char argv[...][...])* for the tokenized ones.

It is possible to process a whole text file as a sequence of commands, with *Dez3DEngine::executeScriptFile(char* fileName)* .

In facts, in *UGP* this method is invoked at startup, and the commands create the initial program environment.

The following is a commented fragment of such file :

```
...
...
#enables openGl culling
drawer set culling 1

#assigns the first texture index to data/textures/white.bmp
drawer load data/textures/white.bmp

#creates a sphere
meshcreator sphere radius 10 parallels 5 meridians 5 in 0 0 -10 color misc

#sets a first person shooter-like camera
camera fps

#sets a free moving camera
camera euler

#sets initial camera position
camera set pos 16 16 7
...
#spawns a 2d entity
new 2d

#sends a message to the entity
msg 2d load data/textures/smiley.bmp
...
```

Chapter 8

The rendering process

8.1 Camera

The various computations performed by a visualization system, can be organized in a sequence of stages, called conventionally *graphics pipeline*.

A typical graphics pipeline consists of:

1. transforming the objects from local to world coordinate space (modeling transformation)
2. transforming the scene from world coordinate space to view space (view transformation)
3. transforming the scene from view space to 3D screen space
4. displaying the resulting rasterized projection

Since **OpenGL** is a rendering library, only the first stage has to be executed by the *UGP* software. The remaining three are executed by **OpenGL**, once the proper commands are given to it. These commands specify the world geometry in world coordinate space, and all of the complementary options to be applied (colouring, texturing, shading, lightning, ecc..) during the rendering process.

8.1.1 The camera and the viewing system

One of the parameters to be given to the **OpenGL** subsystem is the *viewing system*, composed, accordingly to the literature [WATT], by:

1. a view point
2. a view coordinate system
3. a view plane, containing the projection of the viewed scene
4. a view frustum, or volume containing the field of view

In *UGP*, the appropriate solution was to group the code concerning this in a class, DezCamera.

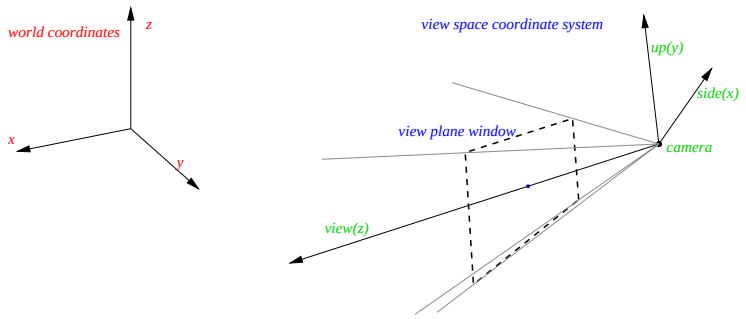

Figure 8.1: World and view coordinate systems.

Such a class contains the routines to hold, modify, keep consistent the viewing system data, and communicate it to **OpenGL** when necessary, to obtain the desired changes.

During the program execution, only one camera object at a time is active, and receives update commands from the engine. These commands are initially captured by the engine input system, interpreted, and sent to the camera.

Since inheritance is possible on this class, it is possible to obtain benefits from polymorphism.

For example, a camera can be set to have constraints, like a limited amount of rotation from a 'base' view vector (like the eyes do with respect to the head), or constraints on the rotation axis. Another possibility is to modify its behaviour in time (e.g.: to continue proceeding forward, slowing, after a key is unpressed, giving a gliding effect, like when navigating liquid surfaces), and thus implement some simulative logic into it.

8.1.2 World to view space transformation

The view space coordinate system considers the camera (or point of view, or eye) as the origin. The one described next is a basic one, similar to the one used in the **OpenGL** library, although more complete (and complicated) viewing systems exist, as the one used in *PHIGS*[WATT] standard.

Conventionally, the *view* direction (the one pointed by the eye) is considered as z axis, the *up* direction as y, and the *side* direction as x (right).

The transformation required to obtain the view space coordinates from the world space is called view transformation, and is represented by the matrix T_{view}.

The T_{view} matrix is further obtainable as the product of a translation matrix T with a rotation one R. Thus,

$$\begin{pmatrix} x_{view} \\ y_{view} \\ z_{view} \\ 1 \end{pmatrix} = \mathbf{T_{view}} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix}$$

where

$$\mathbf{T_{view}} = \mathbf{RT}$$

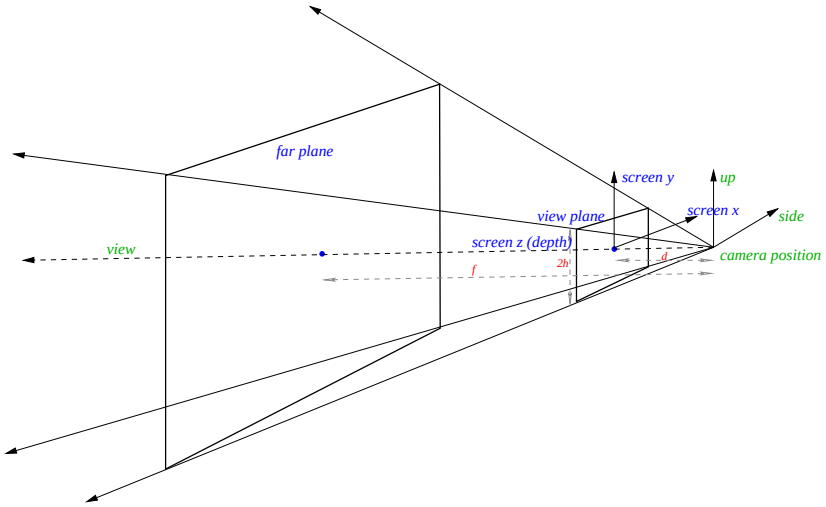

Figure 8.2: The *view frustum* enclosed by the *far* and *view* plane.

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & -camera_x \\ 0 & 1 & 0 & -camera_y \\ 0 & 0 & 1 & -camera_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} side_x & side_y & side_z & 0 \\ up_x & up_y & up_z & 0 \\ view_x & view_y & view_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The view, up, side, and camera position vectors are modified by the camera object accordingly to user input and camera desired behaviour (even animation is possible).

8.1.3 Perspective transformations and 3D screen space

When transforming the geometry into the view space, it is necessary to transform it in a way that perspective effect is given. The farther the objects are, the smaller they must appear. The whole visible space is contained in a frustum (not an infinite pyramid, because of physical hardware limitations). The viewer eye is located at the vertex of the frustum correspondent pyramid, whereas the nearest frustum plane represents the screen on which the image has to be projected, and the bottom represents the farthest location visible.

Moreover, since the perspective transformation projects the view space coordinates to 3D screen coordinates, it should also care about the framebuffer resolution and, generally, about the graphics hardware device.

The following transformation maps the scene onto a screen wide and tall h, with a near plane distance d and far plane distance f:

$$\begin{pmatrix} x_{screen} \\ y_{screen} \\ z_{screen} \\ 1 \end{pmatrix} = \mathbf{T}_{\mathbf{perspective}} \begin{pmatrix} x_{view} \\ y_{view} \\ z_{view} \\ 1 \end{pmatrix}$$

$$\mathbf{T} = \begin{pmatrix} \frac{d}{h} & 0 & 0 & 0 \\ 0 & \frac{d}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{f-d} & \frac{-df}{f-d} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The perspective transformation keeps the z (depth) values in the z *buffer*, a special buffer value associated to each pixel, which is used for depth information of the drawn pixels. When a complex scene is rendered, more objects can cover each other partially on the image, and the ones visible are the ones closer to the viewer. Looking at the z value (in screen space, but in view space is similar) of a new rendered pixel, the rendering system can determine its distance compared to the last one rendered. If the new one is closer (lower z value), it overwrites the old one. Otherwise, it is considered as a pixel 'behind' the previously rendered one, and is rejected.

Another important operation carried out in screen space is polygon clipping, i.e.: ignoring the polygons projected outside the screen boundaries (in this case $\pm h$), or drawing them partially when a portion projects outside.

8.1.4 Transformations and OpenGL

Since **OpenGL** keeps the rendering pipeline data as its state, we must use its API to update it. Unfortunately, the matrices to be passed to **OpenGL** to define the viewing system differs a little from the previously described ones. This because all of the functions to describe the viewing system are several, and during program execution certain are used more often than others. So, according to the **OpenGL** specification, **OpenGL** associates a negative z axis to the depth of the frustum, and thus the view transformation matrix looks like:

$$\mathbf{T}_{\mathbf{view}_{\mathbf{OpenGL}}} = \begin{pmatrix} side_x & side_y & side_z & 0 \\ up_x & up_y & up_z & 0 \\ -view_x & -view_y & -view_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The perspective transformation:

$$\mathbf{T}_{\mathbf{perspective}_{\mathbf{OpenGL}}} = \begin{pmatrix} \frac{h}{aspect} & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & \frac{f+d}{d-f} & -1 \\ 0 & 0 & 2 \cdot \frac{fd}{d-f} & 0 \end{pmatrix}, where$$

$aspect := \frac{screenWidth}{screenHeight}$

$h := cotangent \frac{fov_{width}}{2}$

8.1.5 Two camera classes

For *UGP*, two camera classes where developed, *DezCamera* and *DezGLQuakeCamera*, where the second one is a derived of the first one.

DezCamera is a free move camera with no orientation limits, and it has any bond with the world coordinate system. It allows free rotation around its three axis (view, up and side), and strafing along them. This can result in a somewhat counter-intuitive way of moving: for example, if after veering left (rotating around the view axis) one wants to turn right, the camera view vector will rotate towards the right direction, in this case in the world z direction.

However, the camera simulates free space moving (like in open space or flight).

To provide a more 'earthly' way of moving, *DezGLQuakeCamera* behaves similar to when moving on the earth, turning left and right rotating around the world z axis, but proceeding forward in the view direction, like before. Also, a 'go up' command will move the camera along the world z axis, instead of the camera up axis. It is the camera used in the first person view simulators, and it is based on two angles, θ as azimuth and ϕ as latitude (or elevation angle).

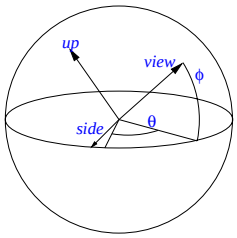

Figure 8.3: Spherical coordinate system.

In this way,
$view.x = cos\phi \cdot cos\theta$
$view.y = cos\phi \cdot sin\theta$
$view.z = sin\phi$
$side.x = -cos(\theta + \frac{\pi}{2}) = sin\theta$
$side.y = -sin(\theta + \frac{\pi}{2}) = -cos\theta$
$side.z = 0$
$up = side \times view$

Additionally, a *roll* angle is used to rotate the up and side vectors around the view vector, allowing to set the camera in all the positions the free move camera allows, but in an easier way.

8.1.6 Communication with the engine

The engine communicates with the camera in a unidirectional way, with three methods:

DezCamera::setStatus(···)
DezCamera::updateStatus(void), and
DezCamera::lookAt(void)

The first method gives the camera the information about what to do in the current time frame. The information is passed in form of a vectors of boolean conditions corresponding to the elementary actions the camera can do (*move forward, backwards, sideways, turn left, veer, etc...*).

The information is stored in the camera, and not executed until the call of updateStatus(). The advantage of this strategy is the ease of logging of

the camera actions (by saving the individual action vectors). This allow for the 'recording' of the camera, for an eventual latter reproduction. However, the solely storage of the action vectors is not enough to obtain a correctly timed reproduction. This because the time frames have different length, not predictable (the computer can execute other processes while executing the program, and other factors make the current load unpredictable). So, to give the user the illusion of constant moving or angular speed, the camera, at each frame reads the time interval from the previous one, and perform its rotations or moves proportionally to the interval length.

For a correct reproduction of a recorded sequence of actions, also the frame interval lengths have to be recorded, and then the behaviour interpolated. This is a trickier task, since the action vector is boolean, and some kind of integration should be done 'reproducing' the actions. The alternative solution is to store the camera position and view vectors data, but it is much less flexible than the previous.

The second method, *DezCamera::updateStatus()*, is called when the state is passed with the previous method, and the camera should calculate the updated values of its view system values. If called two times, the camera executes two times the last actions (for example, if the command of rotating left was given, the camera will continue rotating at a constant speed), until a setStatus(..) call will change its desired behaviour (to stop it, for example).

Once the correct values for the viewing system are computed, they are ready to be given to the rendering subsystem (**OpenGL**), and this occurs during rendering time, in the Dez3DEngine::renderScene() method, with a call to DezCamera::lookAt().

This method passes to **OpenGL** the following parameters: the camera position, view and up vectors. These are the parameters which are most likely to change during the software execution, and can be passed directly (through the use of the *gluLookAt function*), or indirectly, via the creation of the appropriate matrix, and setting it as the current view transformation matrix.

Other parameters, like field of view (in radiants), minimum/maximum depth of view, are modified with other methods in DezCamera methods, and made effective when the perspective matrix is updated. This can be done directly (via a gluFrustum call) or by the creation of the appropriate matrix and applying its effect with a glMultMatrix() function call.

In both cases (*modelview* and *projection* matrix), the update was implemented with the creation/update of the appropriate matrix and applying it on the **OpenGL** matrix stack [WRIGHT].

Once the projection and modelview matrices are correctly loaded, the geometry drawing commands can be executed.

8.2 2D Drawer

A Dez2DDrawer class object is responsible for drawing on-screen 2D graphics. It makes use of **OpenGL** functions to write directly to the framebuffer device.

It is used for drawing images or lines on the top of the rendered 3D geometry. To work correctly, the **OpenGL** two dimensional functions must be called with the proper projection matrix already loaded.

The indicated projection for screen two dimensional drawing is orthographic projection The corresponding matrix is loadable (once the previous one was saved on the *matrix stack* with glPushMatrix) with a call to gluOrtho2D, which creates a matrix like the next one:

$$\mathbf{T}_{\mathbf{ortho2D}_{\mathbf{OpenGL}}} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where r, l denote coordinates for *right* and $left$ clipping planes,
t, b denote coordinates for *top* and *bottom* clipping planes,
f, n denote far and *near* plane distances (recall figure 8.2).
the f and n parameters are respectively 1 and -1, because the *z buffer* algorithm is ignored in 2D drawing. If glOrtho function was called (with different parameters, of course), the f and n were adjustable to specify a proper orthographic projection of the whole scene.

The Dez2DDrawer class makes use of another class, DezGLFontDrawer, which encloses on-screen font drawing functionalities.

8.2.1 Text drawing

The on screen drawing of text is a task done by a DezGLFontDrawer object, member of Dez2DDrawer class.

Such an object has a set of methods for drawing strings on screen, in the desired position and with the desired pixelmap font. Once created, an object is bound to one type of font, which is contained in files located in the *data/fonts* directory. The font name is indicated in the constructor method. At construction time, the object will base on that name to form the names of the 256 files corresponding to the character symbols. If the given filename was "courierNew", it will load into memory files with names "courierNew000.bmp", "courierNew001.bmp", ..., "courierNew255.bmp".

Once loaded, each pixelmap drawing call is associated to an **OpenGL** *display list*, in order to speed up subsequent calls. The display lists are generated once with the glGenLists(256) command,which returns the offset to the call list correspondent to each character.

When a string drawing method is invoked, it usually calls a sequence of commands like:
glRasterPos2i(currentXPos,currentYPos),
glCallList(string[currentChar]+fontListOffset).

The different class methods simply differ in the values *currentXPos*, *currentYPos*, based on the desired text position and spacing.

These are:
*drawStringFromTopWithIndent(const char *s)const*,
*drawString(const char *s,const int xpos,const int ypos)const*,
drawNumber(double n,const int xpos = 0,const int ypos = 0)const,
*drawStringFromBottomWithIndent(const char *s)const*,
*drawStringFromTopWithIndentMultiLine(const char *s)const*

All these methods are public and should be called with the proper (orthographic) projection matrix loaded, as described in the previous section.

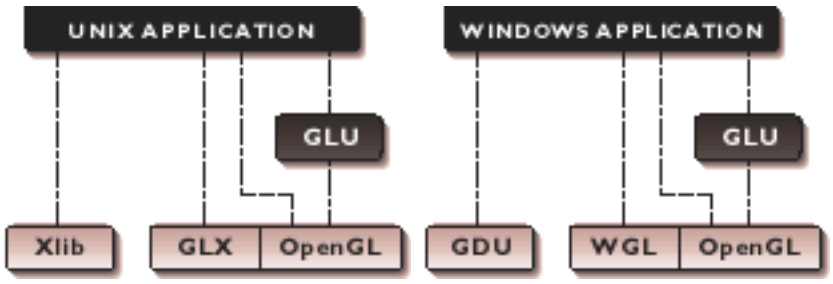

Figure 8.4: OpenGL API hierarchy in Unix and Windows *(from www.sgi.com)*.

8.3 3D Drawer and geometry drawing

The object responsible for on-screen drawing of the geometry is a *Dez3DDrawer* class member of the engine. Its methods encapsulate all the drawing code, which is based on the C routines provided by the **OpenGL** library.

UGP was intended to be as much cross-platform as possible, so the use of OS specific resources is kept at the minimum (none). However, the **OpenGL** subsystem needs to be initialized before use.

8.3.1 OpenGL initialization and portability

The **OpenGL** subsystem initialization is a strictly OS API dependent sequence. This means that in order to be, effective the **OpenGL** routines, the window managing software has to create a proper *drawing context* (data structures mediating access to the graphics device through the graphics device driver and the windows manager).

The problem could be resolved by using conditional compilation in the initialization section, but it would require the knowledge of the underlying window systems (like X and Windows, for example). Besides initialization, other conditional compilation sections would be necessary, for example, in the window resizing code, buffer swapping, input handling and so on... in all tasks where the window manager is usually involved.

The adopted solution to all of these problems, was to employ an intermediate library as an interface between the **OpenGL** and the window management system.

The employed library is called SDL (Simple DirectMedia Library), is written in C, and is distributed accordingly to the GPL*Gnu Public License*.

The SDL library is used for video initialization, input, and buffer swapping. Its use in the implementation is described in 7.2;

8.3.2 Setting up OpenGL state

Assuming that a rendering context was created (as described in 7.2), before the *Dez3DDrawer* object, the **OpenGL** state can be set accordingly to various drawing options.

In the *initOpenGL()* method, called at initialization or whenever the drawing status has to be reset, the two main **OpenGL** matrices (*modelview* and *projection*)

are set to identity and subsequently multiplied for the view transformation matrix the first, and for the projection matrix the second.

This happens in the *DezCamera* code, at every frame for the view matrix (the camera is assumed to be animated), and whenever needed for the projection matrix (its parameters are less likely to change).

Other **OpenGL** status setting commands set/unset fog effect, texturing, depth testing, face culling. All of these options can be switched on/off with a proper glEnable/glDisable call, and some other functions. Drawing options are usually allowed to change at run time with console command options. These are described in the sample initialization file provided with *UGP*.

A fundamental initialization task is texture loading. For the maximum flexibility, textures are loaded when introducing in the program an object using them. Practically, *Dez3DDrawer* receives a *loadTexture(filename)* call, and loads the texture specified in the *filename* file, if not already present. The method returns an identifying index used for polygon texture display. The calling code should use the returned number and assign it to the polygons having that texture on.

When the drawer loads the texture, it uses a technique offered by *opengl*, called *calling list*. Calling lists can help speeding up code execution since declare operations that are likely to be replicated during rendering. Considering that the program should declare to **OpenGL** every texture used by every polygon with a proper function call, a speed up can be fundamental.

A call list is a sequence of **OpenGL** commands executed once and memorized by **OpenGL**. The sequence is identified witn a number. The next time that command sequence is needed, the program will communicate **OpenGL** only that number in a call to *glCallList(...)*. So, every time a texture has to be specified before polygon drawing, the proper call list for that texture is called, and, accordingly to **OpenGL** specification, if the **OpenGL** implementation is well done, a performance benefit can be achieved.

The call list mechanism is suitable for a lot of **OpenGL** commands; however in *UGP* they are used only for textures. This because call lists are static and if the values of the data structures used when creating the list are modified, the list performs the operations as the previous values were still valid. Since a call list update at every geometry update would complicate a lot the code, besides of tying it to the **OpenGL** rendering library, call lists are used only for texture drawing and pixelmap displaying (in the two dimensional drawer class).

8.3.3 OpenGL drawing commands

UGP offers four drawing modes: *vertex mode*, *edge mode*, and textured/untextured *triangle modes*.

The general drawing commands in **OpenGL** should be placed in the portion of code between a *glBegin(GL_enum mode)* and a *glEnd()* call. The *mode* specifies the geometrical object to draw (an edge, lone vertices, triangles, triangle strips, and so on...). The commands appearing in that code could specify vertex colour, texture coordinate, lightning options, and position.

For example,
glBegin(GL_TRIANGLES)
glTexCoord2f(0.0 , 1.0)

glTexImage2D(GL_TEXTURE_2D,0,3,256,256,0,GL_RGB,GL_UNSIGNED_BYTE,-aTexture)

glColor3ub(255,255,255)

glVertex3d(0.0 , 0.0 , 1.0)

...

glEnd()

specifies the drawing of a textured and coloured triangle (although only the first vertex specifying command is shown, another two are necessary).

The methods responsible for drawing the geometry are *drawTexturedTriangles-(), drawTriangles(), drawEdges(), drawVertexes()*, and the analogous *drawMesh-TexturedTriangles(Mesh*m), drawMeshTriangles(Mesh*m), drawMeshEdges(Mesh*m)-, drawMeshVertexes(Mesh*m)*, which differ from the previous ones in the fact they do not access to internal engine structures, and draw only one mesh object a time. The forementioned drawing methods are not directly accessible by the other classes (the methods are declared *protected*), because this would mean less central control from the engine. Thus, depending on the current drawing mode, the current drawing methods are used. They are accessed through the *render-Scene() and renderMesh(Mesh*m)* methods, which contain a function pointer set to the proper drawer method.

Since the *renderScene()* method is invoked by *Dez3DEngine::renderScene()*, and all rendering options are managed by the *Dez3DDrawer* class object, the *Dez3DEngine* has the minimum information on the graphical options (only the video mode, because it has to be set before **OpenGL** is available).

Chapter 9

Entities

Entities are intended to be 'custom' class objects , capable of having various, customized behaviour. For example, *UGP* uses a tissue-like entity, *Dez3DTissue* to simulate skin deformation and needle penetration. Another entity, *DezRobotArmDH*, behaves like a manipulator, controlled by arrow keys.

9.1 *Dez3DEntity*, a common base class for entities

The entity classes are derived from the *Dez3DEntity* class, an abstract class which offers the basic interface common to all of them.

The interface specifies the methods the engine will call to use the entity, and the service methods every entity will be able to use.

9.1.1 Entity flags

At construction time, each entity is required to set its *flags*, which specify its behaviour and desired services from the engine. As soon as the engine adds an entity in its entity container, the flags are read and the engine decides how to make use of the entity.

The *flag* mechanism is implemented as a bitfield in the *Dez3DEntity* class.

The following flag values are possible, and could (should) be combined:

1. *UPDATEABLE*, the state of the entity has to change at every frame;
2. *TAKESINPUT*, the entity reads the user input and behaves accordingly;
3. *DRAWABLE*, the entity geometry has to be drawn at each frame;
4. *DRAWABLE2D*, when the entity is selected, its text/pixelmaps are drawn at each frame;

According to the flags, the engine stores a pointer to the entity into an appropriate list, and each frame, following a fixed sequence defined by the engine (in the base class), certain entity methods are executed.These methods are associated to the declared flags.

9.1.2 Entity geometry and text drawing

In accordance to *DRAWABLE* flag, the virtual draw() method is called when the engine renders the geometry (in the *Dez3DEngine::renderScene()* method).

If not redefined, the *Dez3DEntity::draw()* method is called, and by default draws the contents of the *Dez3DEntity::meshData* array, if any.

Otherwise, an entity can place custom drawing code into the draw() method, but it is a more risky operation, since incorrect **OpenGL** operations could influence subsequent drawing instructions.

The draw() method is declared friend of the *Dez3DDrawer* class, so it can refer directly to its drawing methods concerning meshes, edges, points, and the geometry will be drawn accordingly to the current drawer settings.

The *DRAWABLE2D* flag specifies a slightly different behaviour, since the correspondent commands draw rendered text and pixelmaps directly into the frame buffer, on the top of the rendered geometry, and are executed only if the entity is currently *selected*.

The corresponding interface methods are draw2D() and printBuffers(). They are also called in *Dez3DEngine::renderScene()*, after the geometry is loaded and the orthographic projection matrix is loaded in the **OpenGL** matrix stack.

The *draw2D()* method is declared friend of the drawing classes, so it can refer directly to the drawing methods concerning text and pixelmaps.

9.1.3 Entity update and input

The *UPDATEABLE* flag informs the engine to call the virtual performAction() method on that entity at every frame, from the virtual *Dez3DEngine::update()* abstract method.

Finally, the *TAKESINPUT* flag informs the engine that the entity can accept user input, in form of information on the pressed keys. However, if the read input code was in the entity, it had to contain system specific calls or SDL (see sections 7.2,6.1) calls, and this would break the clarity rules, aside from creating a chaotic situation prone to bugs.

So the choice made was to leave the input capture task to the proper engine derived class (like SDLEngine), and to send it, when necessary, to the entity. In order to receive user input, the entity must be *selected* as *current* by the engine, and the engine must be in *entity mode*. The engine entity mode is obtainable when the user selects the desired entity and gives the proper engine command to 'control' it.

The key combination to quit the entity mode is recognized by the engine, so there is no chance for the user to get stuck with a bad programmed entity.

In the user commands section there is the list with the aforementioned commands.

9.2 Some examples with pictures

This section introduces to some of the *entities* instantiable in *UGP*.

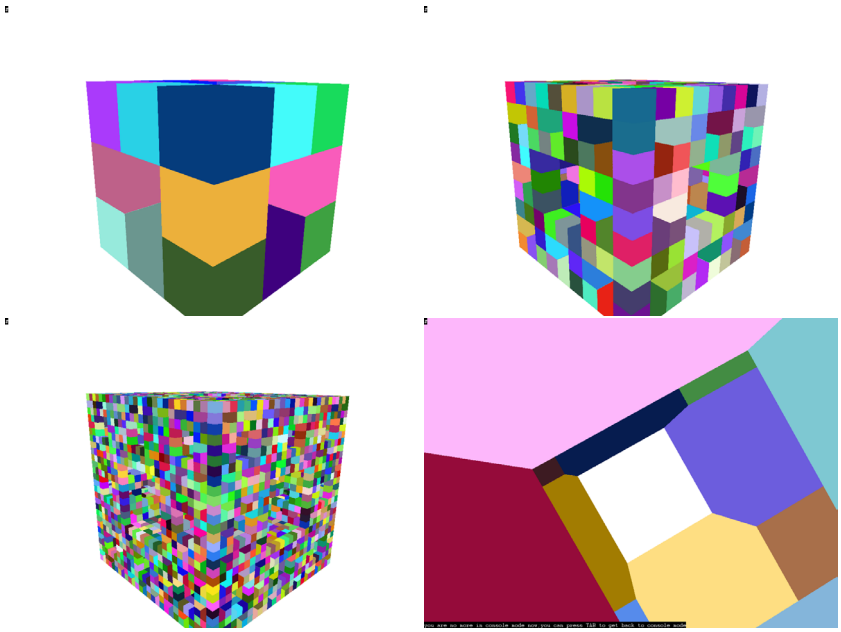

Figure 9.1: *DezMenger* objects, representing Menger sponges of order 1,2,3, and through the axial 'hole' of an order 1 one.

9.2.1 Menger sponge

The Menger Sponge, on figure 9.1 is a fractal construction, obtained, at each iteration, from a solid cube by first splitting it in 27 cubes, each one wide one third of the original. The 7 axial (on the 3 axis) cubes are removed from the original 27, thus 'carving' axially in it. At each iteration of 'carving', each of the 20 resulting cubes is split and 'carved', in turn.

An intersting fact about this solid is that, given V_0 and A_0 as the volume and surface area of an initial cube, the following formulas stay for the i^{th} iteration of the 'carving' process:

$$V_i = \left(\frac{20}{27}\right)^i V_0 \Rightarrow \lim_{i\to\infty} V_i = 0$$

$$A_i = \left(\frac{4}{3}\right)^i A_0 \Rightarrow \lim_{i\to\infty} A_i = \infty$$

It is troublesome to imagine a solid with infinite surface and with no volume!

The command to be given for a menger sponge object is *new menger 3*.

9.2.2 Bouncing sphere

It is also possible to see a sphere bouncing on the $z = 0$ plane. The spawn command is *new bouncing*.

Figure 9.2 gives an example.

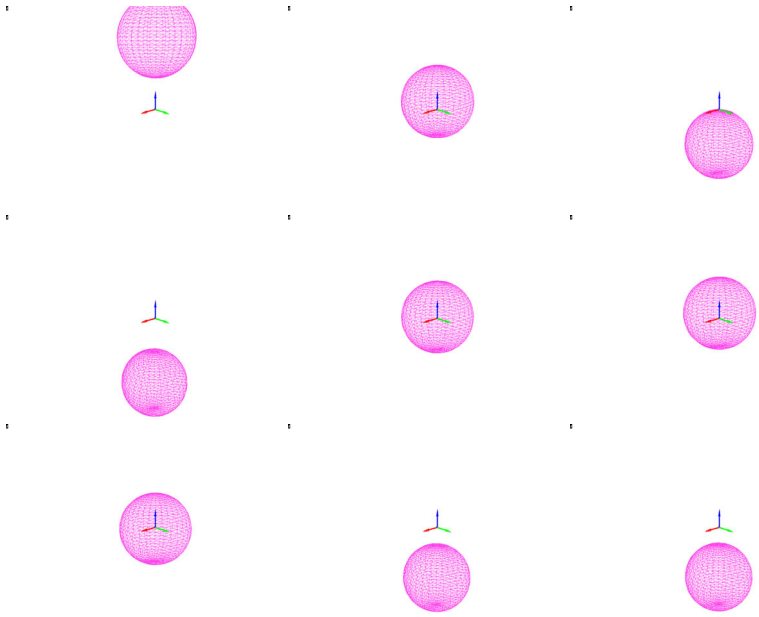

Figure 9.2: A bouncing ball.

9.2.3 Shootable

Typing the command *new shootable*, a sphere is spawned, and can be used as a target for 'shooting'. When the user is nearby and, once selected it presses *s*, the sphere changes colour if hit as shown on figure 9.3.

9.2.4 Tissue

Already described informally in section 3.3.1, the *DezTissue* class represents a systolic mesh modifiable by a user controlled *needle*.

The spawn command is *new tissue TEXTURE*, where *TEXTURE* is the filename, or the number of the preferred 'skin' texture [1].

Once selected the object, the tissue needle is controllable with the following keys:

- UP ARROW pushes the needle along x
- DOWN ARROW pushes the needle along $-x$
- RIGHT ARROW pushes the needle along y
- LEFT ARROW pushes the needle along $-y$
- S pushes the needle along z
- X pushes the needle along $-z$
- P pushes the needle deeper
- R releases the needle

[1] For more information on texture support, see section 8.3.2.

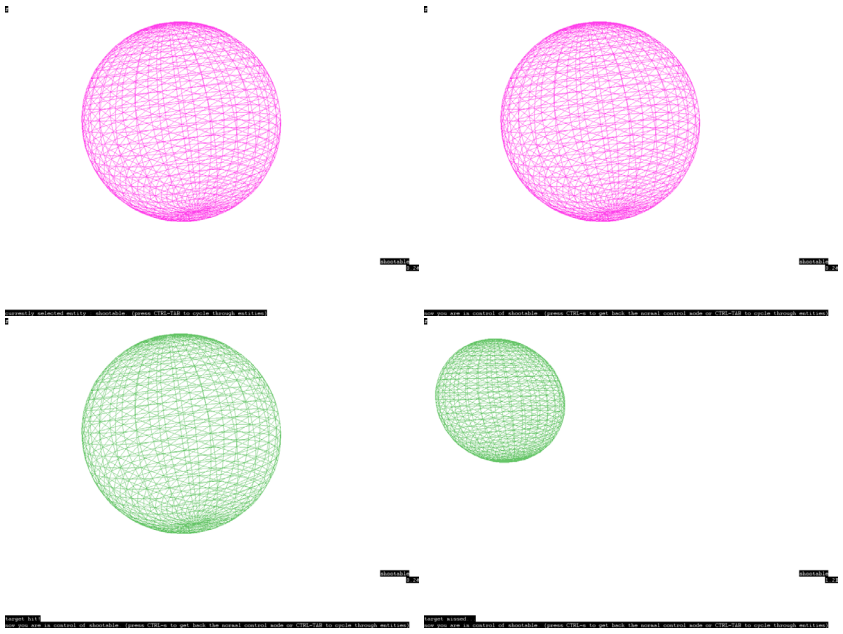

Figure 9.3: A *DezShootable* class object : to be selected, selected, hit, missed.

H displays some help
M the needle penetrates faster, when keys are pressed
L the needle penetrates slower, when keys are pressed
Figure 3.3 shows a instantiated *DezTissue* object.

9.2.5 The robot arm

Already described informally in section 3.2.1, the *Dez3dDezRobotArmDH* class represents a multi link manipulator.

The constructive parameters are a Denavit-Hartenberg table *theta,d,l,alpha* and some strings, contained in some file, structured as the following:

```
{
 name popeye
 linksCount 4
}
#link # theta d l alpha
{
 link  1 0.3 3 2 0
 meshcreator orientedcylinder .1
}
{
 link 2 1.6 0 1 3.1414
 meshcreator orientedcylinder .1
}
{
 link 3 0 1 0 0
```

```
 meshcreator orientedcylinder .1
}
{
 link 4 1 .0003 0.5 0
 meshcreator orientedcylinder .1
}
```

The above text encodes a manipulator looking as the one in figure 3.2.
The spawn command is *new robotfile FILE*, where *FILE* contains the above text.
The control commands are:
LEFT ARROW the current link increases its *theta* value.
RIGHT ARROW the current link decreases its *theta* value.
TAB changes the currently selected link

9.2.6 2D Image Displayer

The 2D image processing procedures described in sections 4.3, 4.4, were implemented in an entity class called *Dez3DImageDisplayer*.

The following screenshots were taken by first spawning a *Dez3DImageDisplayer* class object, and then giving specific commands. Figure 9.4 shows the sample image and its

Figure 9.4: Our sample image and its histogram.

histograms, displayed in *UGP*, issuing the commands:

```
msg 2d load babcia.bmp
msg 2d --drawHistogram screenshotBabciahistogram.bmp
```

The images in 9.5 were obtained, respectively, issuing two times the first command and one time the second command:

```
msg 2d --blur 16 0 0
msg 2d --edges 0.14
```

The resized images in 9.6 were obtained with:

```
msg 2d --resize 360 700
msg 2d --resize 3000 1400
```

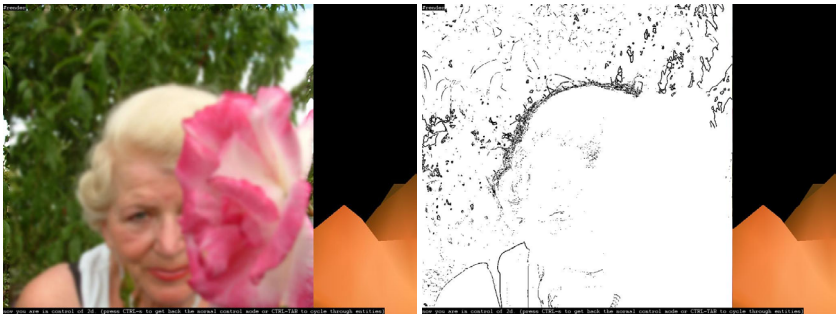

Figure 9.5: Blur and edge revealing operations.

Figure 9.6: Resizing of images in *UGP*.

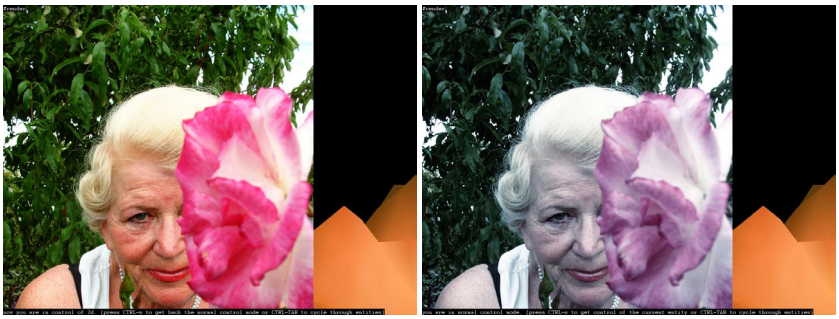

Figure 9.7: Contrast enhancing and the concatenation of partial desaturation,contrast and saturation.

The left image in 9.7 enhances the contrast of the original image, with the command:

```
msg 2d --contrast .9
```

The right one was obtained with the combination:

```
msg 2d --desaturate 0.8
msg 2d --contrast 1
msg 2d --saturation 1
```

Figure 9.8 shows the box-pixelated version of our image:

```
msg 2d --boxPixelate 10 10 3 3 2
```

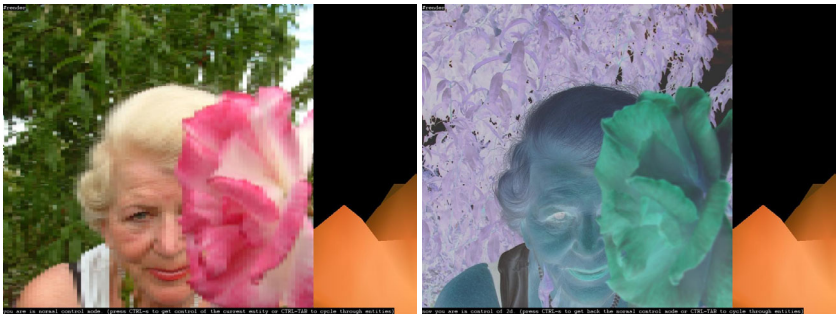

Figure 9.8: The sample image after box pixelating and negative effects.

and a 20% blend of the original image with the negative version.

```
msg 2d --negative 0.8
```

Chapter 10

MeshCreator class as a shape factory

10.1 A primitive shape creator

The *Mesh* class has a great deal of opportunity of representing diverse triangle meshes, but there is the problem of creating and specifying meshes!

The employed multiple representation (triangles, edges, vertices), is practical for rendering purposes, but as any triangle mesh representation suffers of the lack of creation ease. In facts, the simplest mesh available, the tetrahedron, has the following pointer schema :

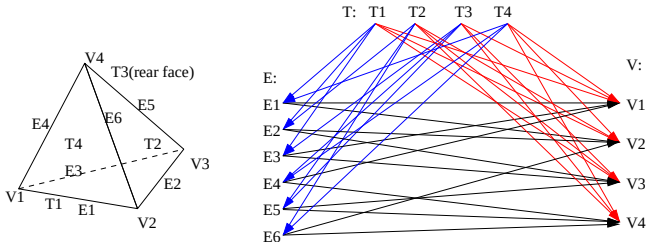

Figure 10.1: Structure of pointer interconnections for a tetrahedron.

There are totally 48 interconnections, for a structure with only 4 vertices!

Being evident that manual editing of meshes a difficult task, the alternative solution for the creation of more complicated meshes is the aggregation (or even merging, altough not implemented) of primitive shape meshes.

So, the primitive shapes are created once in the methods of a *MeshCreator* class object, and modified as needed. For this, the *stateless MeshCreator* class offers a method accepting string commands for the creation of meshes. The string is parsed then, and the appropriate methods of *MeshCreator* are invoked for the creation and modifying of the desired mesh.

10.2 The *MeshCreator* command string

Recognized commands have the format :

SHAPE PARLIST [in X Y Z] [bounded XBMIN YBMIN ZBMIN XBMAX YBMAX ZBMAX] [rotated XR YR ZR] [color <R G B> | random | misc] [reversed] [texture TEXTUREINDEX]

The bracketed arguments are optional, and can appear in any order. The lowercase words are keywords identifying each argument option.The angular brackets group the contained text.

The replacement rules for the variables (uppercase text) are:

SHAPE:= cube | sphere | arrow | flatSystolicMesh | terrain | disc | leaf | cylinder | moebiusdisc | box

X,Y,Z,XR,YR,ZR,XBMIN,YBMIN,ZBMIN,XBMAX,YBMAX,ZBMAX:= a real

TEXTUREINDEX:= a positive integer, or a file path[1].

R,G,B:= a real, in the [0,1] interval

The *PARLIST* string is specific to each SHAPE, and is as follows:

for *arrow : XDIR YDIR XDIR*

for *sphere : [radius R] [parallels P] [meridians M]*

for *flatSystolicMesh : [cellSize CS] [xCells XC] [yCells YC]*

for *flatSystolicMesh : [cellSize CS] [xCells XC] [yCells YC]*

for *terrain : [cellSize CS] [xCells XC] [yCells YC] heightmap HFILENAME colormap CFILENAME*

for *cylinder : [verticalSubdivisions VS] [circularSubdivisions CS] [radius R] [height H]*

for *moebiusdisc : [radialSubdivisions RS] [circularSubdivisions CS] [minRadius MINR] [maxRadius MAXR]*

for *moebiusdisc : [radialSubdivisions RS] [circularSubdivisions CS] [minRadius MINR] [maxRadius MAXR]*

for *box : XMIN YMIN ZMIN XMAX YMAX ZMAX*

for *orientedcylinder : XMIN YMIN ZMIN XMAX YMAX ZMAX RADIUS*

where radius and position information are real numbers, all other are integers, except string filenames.

[1] Altough in this case the texture name string will be read and interpreted by the engine, because *MeshCreator* accepts only an integer as argument (this because *MeshCreator* has not texture managing functions really).

Chapter 11

UGP usage

*UGP*has three interactive operating modes: *normal*, *console*, and *entity*. The first is used to move around in the three dimensional environment. The second is used to enter and execute textual console commands. The third is used to control interactively an entity.

11.1 Mode switching commands

Tab	NORMAL MODE $\Longleftrightarrow$ CONSOLE MODE
ESCAPE	ANY MODE $\Longrightarrow$ NORMAL MODE
CTRL S	NORMAL MODE $\Longleftrightarrow$ ENTITY MODE (current entity, if any)
CTRL TAB	Cycle through the entities, to select the *current*

11.2 Normal mode commands

A	Roll left
D	Roll right
S	Move upwards
X	Move downwards
Z	Strafe left
C	Strafe right
CTRL P	Pause
H	Help
Q	Stabilize camera view
R	Reset camera view
M	Toggle mouse
F	show FPS
F1	Toggle texture drawing
F2	Increase camera FOV

F3 Decrease camera FOV
F5 Increase camera moving velocity
F6 Decrease camera moving velocity
F7 Increase camera rotation speed
F8 Decrease camera rotation speed
F9 Toggle fog
F10 Toggle vertex/edge drawing
F11 Toggle edge/triangle drawing
F12 Screenshot in the current directory (bmp format)
UP ARROW | MOUSE RIGHT BUTTON Move forward
DOWN ARROW | MOUSE LEFT BUTTON Move backward
LEFT ARROW | MOUSE LEFT Move left
RIGHT ARROW | MOUSE RIGHT Move right
PAGE UP | MOUSE DOWN Look up
PAGE DOWN | MOUSE UP Look down
CTRL TAB Cycle through the current entities
CTRL S Select the current entity (going into ENTITY MODE)
CTRL Q Quit the program
CTRL P Pause the program
CTRL R Reset the program

11.3 Console mode commands

ENTER Submit command
UP ARROW Previously executed command
DOWN ARROW Next executed command
CTRL C Copy command buffer
CTRL X Cut command buffer
CTRL V Paste command buffer
CTRL U Clear command buffer
BACKSPACE Clear last character in command buffer
Escape | TAB Exit command console mode
ANY CHARACTER KEY Add typed charater to command buffer

Bibliography

[WATT] Alan Watt. 3D Computer Graphics, third edition. Addison Wesley 2000

[WRIGHT] Richard S.Wright Jr,Michael Sweet. OpenGL. Waite Group Press 1999 (Polish edition by Helion, 1999)

[SGI] various authors. OpenGL Programming Guide (better known as Red Book). www.sgi.com

[SDL] Sam Lantiga. SDL (Simple DirectMedia Layer) User Guide. http://www.libsdl-.org/

[LGP] John Hall. Programming Linux Games. http://www.overcode.net/ overcode-/writing/plg 2001

[FOLEY] J.D.Foley Et Al. Introduction to Computer Graphics. Addison Wesley Long-man 1994

[ZONNEV] F.W.Zonneweld. Digital visualization of the medical reality : a glimpse of the future. http://www.medical.philips.com/main/news/assets/docs/medicamundi-/mm_vol40_no2/zonnev.pdf

[GPG1] Dante Treglia. Game Programing Gems 1.Charles River Media 2001 (Polish edition by Helion, 2002)

[GPG2] Dante Treglia. Game Programing Gems 2.Charles River Media 2001 (Polish edition by Helion, 2002)

[GPG3] Dante Treglia. Game Programing Gems 3.Charles River Media 2002 (Polish edition by Helion, 2003)

[GT] Various code from http://gametutorials.com

[GD2007] Frdric Patin. An Introduction To Digital Image Processing. http://www-.gamedev.net/reference/articles/article2007.asp

[MCCM] Ada Puglisi. The music of the colour, the colour of the music. http://www.sifi.it-/en/rivista/rubriche/Articolo%20sinestesia/Sinestesia.htm

[KW] Piera Giovanna Tordella. Kandinskij. Elemond Arte, 1992

Index

Dez3DImageDisplayer, 66
Dez3DTissue, 61
Dez3dDezRobotArmDH, 65
DezRobotArmDH, 61
DezTissue, 64
DezCamera, 54
Dez3DCommandConsole, 49
Dez2DPoint class, 44
Dez3DDrawer, 58
Dez3DEngine, 47
Dez3DEntity, 61
Mesh, 69
MeshCreator, 69
Mesh class, 44
Vector3D<X> template class, 44
Vector3D class, 44
VertexData class, 44
OpenGL, 48, 54, 58
OpenGLinitialization, 48
OpenGL library, 45
OpenGL matrices, 54
UGP classes, 43
2D drawer, 56
3D drawer, 58
3D screen space, 53

back face, 35
blackness, 25
blur, 26
bouncing sphere, 63
brightness, 25

camera, 51
chroma, 22
code documentation, 45
colour, 21
colour space, 22
commands, 64
components of *UGP*, 43
containment, 34
contrast, 28
convexity, 38
culling, 35

Denavit-Hartenberg table, 65
desaturation, 24
despeckle, 29
Dez2DDrawer, 57
Dez3DEngine::command, 49
Dez3DEngine::executeScriptFile, 49
DezCamera::lookAt, 55, 56
DezCamera::setStatus, 55
DezCamera::updateStatus, 55
downsampling, 32

edge, 37
engine, 47
entities, 61
entity flag, 61
event, 48

front face, 35

gaussian blur, 26
glBegin, 59
glMultMatrix, 56
gluOrtho2D, 57
graphics pipeline, 51

HSV, 22
hue, 22
hull, 36

image processing, 23, 24, 66
initialization file, 50, 65, 69
input, 48
intersection, 35
inverted edge, 39

kandinsky, 21

logical pointer, 39

matrix convolution, 26
Menger sponge, 63
mesh, 36
mesh data structure, 39
mesh properties, 37
Munsell, 22

negative, 24

non punctual transformations, 26

OpenGL API hierarchy, 58

perspective transformation, 53
plane, 33
point, 33
programming style, 42
punctual transformations, 24

ray, 35
rendering, 51, 56
resampling, 31
resizing, 31
RGB colour space, 22
RGB cube, 23

saturation, 24
SDL, 42, 45, 47, 48
SDL library, 45
SDL_Event, 48
SDL_SetVideoMode, 48
SDL_VideoModeOK, 48
SDL_WM_GrabInput, 48
sdlengine, 47
shapes, 69
sharpen, 30
straight, 33

Tartaglia-Pascal triangle, 28
terrain representation, 14
tetrahedron, 39
text drawing, 57
texture, 70
triangle, 37
triangle splitting, 38

upsampling, 31

view space, 52
viewing system, 51
virtual, 49

whiteness, 25
wireframe, 40