DUBLIN CITY UNIVERSITY

SCHOOL OF ELECTRONIC ENGINEERING

3-D Human Modelling.

Gaelle Rougier M.Eng.

September 2001

MASTER OF ENGINEERING

IN

TELECOMMUNICATIONS SYSTEMS

Supervised by Dr. D. Molloy

DUBLIN CITY UNIVERSITY

SCHOOL OF ELECTRONIC ENGINEERING

Acknowledgements

I would like to thank my supervisor Derek Molloy for his guidance, enthusiasm and commitment to this project. I would also like to thank Robert Sadleir for his help and his advices. Finally I would like to express my appreciation to Derek Molloy for preparing the original Transfer Report template, from which this document is based.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: .... Date: ....

Abstract

This project investigates the creation of virtual human models that can be used in image processing tasks such as golf training systems. For example, a human elbow can only bend in a certain way, similarly for a shoulder, knee, etc. If a human is being tracked in a scene it should be possible to constrain the possible tracking solutions using this rules. If the model can be maintained on the human through the sequence it should be possible to describe the motion that is occurring. This is similar to the concept of Inverse Kinematics (IK) as used in the three-dimension (3-D) graphics simulation of humans.

Before addressing the problem of human modelling, it is essential to understand how the human brain interprets the world to generate that feeling of three-dimensionality, as well as to learn how to implement this in a computer environment. Stereographics equipment has been made available to aid in the 3-D visualisation aspect of that project. It is mainly composed of an Infra Red (IR) emitter and of crystal eyes shutter glasses.

The human model built in this project makes use of the OpenGL graphics library as well as of the C++ programming language. It's implemented using Stereographics hardware 3-D visualisation and it demonstrates the validity of human modelling using the investigated techniques. The final model can be animated, moved and resized by the user using a menu. Other features are also provided (as options related to the adaptation of the 3-D feeling to the particular distance between the eyes of the viewer, increase of the rotation speed, zoom, etc).

Table Of Contents

Acknowledgements iii

Declaration iii

Abstract iv

Table Of Contents v

Table of Figures viii

1. Introduction 1

1.1 Background 1

1.1.1 3-D human modelling 1

1.1.2 3-D human modelling in sports area 2

1.2 Issues addressed in this project 5

1.3 Structure of the dissertation 5

2. Theoretical and Technical Background 8

2.1 Stereoscopic imaging 8

2.1.1 Effect of three-dimensionality 8

2.1.2 Principle of Computer environment 9

2.1.2 Quad buffering scheme 10

2.1.3 Stereoscopic synchronization control 11

2.1.4 Parallax 12

2.1.5 Introduction to Stereoscopic Perspective Projections 12

2.2 Software implementation for the stereo 16

2.2.1 Initialising a Window to enable stereo and testing it 16

2.2.2 Writing to Separate Buffers 18

2.3 Motion control 19

2.3.1 Key framing 20

2.3.2 Geometric motion control 21

2.3.3 Physical motion control ... 22
2.3.4 Behavioural motion control ... 22
2.4 Motion capture ... 23
2.4.1 Magnetic motion capture ... 23
2.4.2 Optical Motion Capture Systems ... 25
3. Design and Implementation ... 29
3.1 First steps ... 29
3.1.1 Building a new project ... 29
3.1.2 Work done in the 3-D World ... 30
3.1.3 Stereo Effects ... 31
3.2 OpenGL basics ... 32
3.2.1 Drawing basics ... 32
3.2.2 Other useful functions ... 33
4. The Frame Bottom Model ... 38
4.1 Building the basic static model ... 38
4.2 Animation and implementation of the model ... 40
4.2.1 Vertical displacement of the body ... 40
4.2.2 Walking animation ... 42
4.3 Class behaviours for the application code ... 44
5. The Complete Human Model ... 45
5.1 Building the static model ... 45
5.1.1 Design of the model ... 45
5.1.2 Hierarchy and drawing of the model ... 49
5.2 Animating the model ... 52
5.2.1 Walking Animation implementation ... 52
5.2.2 Menu to act on the model ... 62
5.3 Testing the walking model ... 64
5.3.1 Coding tests ... 64
5.3.2 Visual tests ... 67

5.4 Sizing the model 71
5.4.1 Adding an item to the menu 72
5.4.2 The Dialog Box 73
5.4.3 Making the Dialog Box work 74
5.4.4 The application Dialog Box 76
5.5 Class behaviours for the application code 78
6. Conclusion 80
6.1 Summary 80
6.1.1 Work done 80
6.1.2 Problems encountered and solved 82
6.1.3 Problems encountered but not solved 85
6.2 Future Work 85
6.2.1 Geometrical improvement 86
6.2.2 Motion control 88
References 94
Appendix 1: How to install the glut library 98

Table of Figures

Figure 1. Virtual football trainer 3
Figure 2. Golfer recorded using sensors 4
Figure 3. Principle of stereovision 9
Figure 4. Crystal Eyes glasses and emitter 9
Figure 5. Double-buffered (left picture) and Quad-buffered (right picture) memory organisation 11
Figure 6. A: Positive parallax (left picture), B: Negative Parallax (right picture) 12
Figure 7. Perception of Parallax Effects 13
Figure 8. Assuming a single camera 14
Figure 9. Offset applied to the whole scene --> ONLY negative parallax 15
Figure 10. Shift of the projection --> negative AND positive parallax 16
Figure 11. Key frames and single frames 20
Figure 12. Motion capture simple marker configuration 26
Figure 13. First Application 31
Figure 14. Bottom model components 38
Figure 15. Bottom model hierarchy 39
Figure 16. Vertical Displacement 41
Figure 17. Angle for the lower vertical leg 41
Figure 18. The “Animated” Bottom Model 43
Figure 19. Human model components 46
Figure 20. Model hierarchy 49
Figure 21. Relations in the model 51
Figure 22. Initial upper leg angles 54
Figure 23. Angles for the body's vertical displacement 57
Figure 24. Rotation of the leg 60
Figure 25. 'File' menu 62
Figure 26. 'Stereo' menu 63
Figure 27. 'Human' menu 63

Figure 28. The “Animated” Human Model 64
Figure 29. Smaller amplitudes 68
Figure 30. Initial Amplitudes.. 69
Figure 31. Abnormal animation of the model.. 70
Figure 32. Real human walking 71
Figure 33. Human model application walking... 71
Figure 34. A model that matches anyone... 77
Figure 35. Non expected rotations 82
Figure 36. Transformation order 83
Figure 37. A: Actual foot, B: Possible improved foot 86
Figure 38.Creating triangle strips 87
Figure 39. Cal 3d demo... 88
Figure 40. Constraints exists in the body... 90
Figure 41. A: Rotation along the Y-axis, B: Rotation along the X-axis............................ 92

Chapter 1

1. Introduction

The three-dimensionality (3-D) effect is a consequence of how the human brain interprets what the eyes see. Being able to reconstruct the sense of 3-D using a computer is a useful visualization tool, which can be performed using stereo viewing. Once it is possible to have a natural 3-D view of an object, it is of great interest to apply this to a human model. Indeed, representing human models has been a major research area in the field of computer science for the past number of years. The ability to view a human model in 3-D gives an extra dimension so that more information can be represented but also understood in one picture, since it increases the ability to interpret the multidimensional data related to the model. All these great features can be applied to sports science in order to train people and help them improve at their favourite sport.

1.1 Background

1.1.1 3-D human modelling

For years, modelling and animation of the human body has been an important research goal in computer graphics. The principal aims of these researches being entertainment, medicine, architecture, urban planning, and virtual reality.

In fact, there are two major difficulties in rendering human animation: motion of the hierarchical structure (or skeleton) and deformation of the body. Animation of the skeleton is discussed in this report. Modelling the deformations of human bodies during the animation is an important but difficult problem, with the work in this area depends on the applications. This last issue is not discussed, since it is not a major concern of the project.

In initial research, humans were represented as simple articulated bodies made of segments and joints with a kinematics based model to simulate them. More recently, dynamic models have been used to improve the continuity and fluidity of the movement. Though virtual human model are in existence, they are mainly used for research purposes to enable the simulation of human movements and behaviours. Traditional character animation techniques are time-consuming tasks without any real-time possibilities. Design and animation of such characters is a time demanding operation as it is done using a general-purpose animation of system like Softimage[1], Alias-Wavefront[2], Maya [18], or 3D StudioMax [17].

Apart from medical applications, these virtual human models can be used to help people to learn their favourite sport.

1.1.2 3-D human modelling in sports area

When they deal with the sport area, training applications are basically done by means of video games. However, the rapid development of virtual reality technologies has allowed great advances and opportunities for team-sport training by investigating the area of virtual simulations.

As an illustration, while video games allow the user to explore a football play by looking at the computer's monitor, immersive virtual reality provides a much more realistic experience and form of training. The Virtual Football Trainer [20] allows the training of football players in the Virtual Reality CAVE (Cave Automatic Virtual Environment), which is currently the most advanced system for immersive virtual reality. The objective of this trainer is to train a player for the correct visual perception of play situations and for the fast reaction to the movements of other players on either team (Figure 1).

---

[1] SoftImage Corporation – http://www.softimage.com/

[2] Alias Corporation – http://www.aliaswavefront.com/

Figure 1. Virtual football trainer

The animated movement of a virtual player is controlled via an internal skeleton. A three-dimensional geometric shell that represents the player's external shape envelops the skeleton. A library of hundreds of pre-defined poses and movements defines the positions and angles for these joints on a time grid.

But these features are not only offered to team sports. Individual sports such as the golf also have their own application in 3D. A golf swing analysis and training system named SkillTec 3D-Golf [21] has recently been developed (year 2000) to provide a method of swing training: the users swings are measured and compared to a database of expert swings (a prescription for rapid improvement is then provided). This golf trainer makes use of a real time motion capture and measurement system that analyses every nuance of human motion in three dimensions. This system has been developed by Skill Technologies (it is primarily used in famous research universities and hospitals all around the world, and via this trainer it's now also available to the golfing community). The data are actually got by placing small sensors on the body of the golfer, on the pelvis, the back, the head and the left hand as shown on Figure 2.

Figure 2. Golfer recorded using sensors

These sensors measure the electromagnetic field emitted by a nearby emitter and convert them to body position and rotation measurements. It allows to precisely measure every movement the golfer makes. The computer displays these movements many times per second. Not only is the movement displayed as a virtual reality three-dimensional model of the golfer, but dynamic parameters of the swing are also computed and saved for analysis. Using this recorded data, it is then possible for the instructor and the golfer to visualise the swing from any angle and any position (back, front, top, bottom, left and right). It is also possible to replay the scene one frame at a time to view errors such as excessive head, hips, or torso motion. The user can also compare its movement to a professional swing available in the database and identify the differences. A free download version of this system allows the user to view 3D animated models of two different golf swings; a teaching professional swing and the swing of a weekend golfer with twenty handicaps: to have access to the entire version of the viewer it is necessary to go for a lesson in SkillTec 3D-Golf Hi-Tech Golf Learning Centre in Phoenix.

1.2 Issues addressed in this project

This report is an investigation of virtual 3-D human modelling. According to this, the main objectives of this project are to become familiar with3-D graphics programming techniques (OpenGL [6-10], C++ [12-16], and Stereographics API. [1-5]), but also to investigate the use of these techniques in human modelling and implement, using Stereographics hardware 3-D visualisation, an application that demonstrates the validity of human modelling using the investigated techniques.

The problems encountered during this project are mainly programming problems, since familiarity with many languages has to be achieved before the start of such a project. Doing tutorials available on the Internet allows this. But everything is not available on tutorials, and as an example it is not that easy to display a dialog box called by a menu and to deal with its content in the case where the working project is not a Microsoft Foundation Class (MFC) application but instead a Win32 application. OpenGL is a very powerful graphics library, but it needs some knowledge to use it). For instance saving the current references before doing any transformation is important, even if it is not obvious at first sight. Smaller problems such as installing new libraries and mapping them to the project are also things that need to be addressed, and no particular tutorial is available for this: it is "trial & error". The animation of the model is something that requires a lot of work as well. Indeed, the model has to be thought from the very start as a model that should then be animated. It is therefore indispensable to build it in a structured way. The animation in itself is something that is not that simple and that has to be investigated during this project.

1.3 Structure of the dissertation

Chapter 1 is an introduction to the project.

The work done in the area of human modelling is overviewed, and the main objectives of this project as well as the structure of the dissertation are specified.

Chapter 2 deals with the theoretical and technical background required for a good understanding of the project.
The feeling of three-dimensionality in the real world is explained, as well as its implementation in a computer environment. It then deals with motion animation of a human skeleton, so that the reader can have an idea of the different existing techniques in that area, but also since the model to be created has to be animated. Since animating a human model generally involves a databank, an explanation of motion capture is provided.

Chapter 3 is mainly dedicated to a further programmer, since it reports the first steps of this project.
In this part is explained how to build a new project as well as the major functions that are assumed to be indispensable to draw a model and animate it. An executable (available on the disk) is done from this part and is available, so that anyone can build on it and test its understanding of the OpenGL functions. This executable contains a pyramid and a quad rotating around two different axes, on which stereo effects (allowing 3-D) are available via the menu. Tests are not included in the code of the executable, since the aim in here is to use the OpenGL functions and practice (anyway these functions can be copied in the final project into which they have been built for security reasons).

Chapter 4 explains how to build a bottom-walking model by having a hierarchical approach of the body.
This bottom is first built in a static way before animation is added to it by taking advantage of the key framing technique explained in Chapter 2. The angles are hard coded and correspond to the key frames angles. A key problem is addressed in this part: it is the displacement of the base according to the leg movement. Indeed, when the legs are bent, the human model should not seem to fly, and its base should follow the movement.
Finally, an explanation of each class is presented.

Chapter 5 details the whole final body construction.

Once again a hierarchical approach is taken, and the animation uses the key-framing concept. Actually the angles are not hard coded anymore, but read from a file. It is therefore possible to change theses angles and animate the model using databanks of angles. Many features (such as sizing the model, adding frames to the animation, animating the model from the menu, rotating it…) are added to the model (the way to do it is explained in detail, and more specifically the way to deal with a dialog box. The model tests are made available to the reader in this chapter: code tests and visual tests are presented, as well as their results. Once again, to conclude, an explanation of each class of the application is presented to allow a better understanding of the application code.

Chapter 6 concludes this document, summarizing the work done and the problems encountered. This chapter also discusses future directions that might be taken by the student continuing this project over the next three years.

In this chapter some interesting background in the area of human modelling is provided, and some existing systems in the sport area are also discussed. Following this, the issues addressed in this project and the structure of this dissertation are then performed. The next chapter is dealing with more technical background (such as the sense of 3-D) that it is necessary to be aware of before addressing the problem of human modelling.

Chapter 2

2. Theoretical and Technical Background

This chapter explains how the human brain interprets information in order to create a feeling of three-dimensionality, and how it is possible to reproduce that feeling in a computer environment, using stereo. A review of the different kind of the existing motions of a human body is presented, including the two main methods for dealing with motion capture, since this project shall investigate the creation of an animated three dimensional human model.

2.1 Stereoscopic imaging

2.1.1 Effect of three-dimensionality

The sense of three-dimensionality results from how human brain processes what the eyes see. The distance between human eyes (on average 64 millimetres apart for adults) causes each eye to see a scene from a slightly different point of view. The brain interprets this information by combining the two different, although similar, images into one, and the resultant sense of depth is known as “stereopsis” [4][5].

This can be illustrated by looking at a simple pyramid (Figure 3): the left and the right eyes, represented by two circles, see the object from positions about six centimetres apart. The two small pyramids represent what the eyes see, and the brain combines these pictures to create a stereo impression.

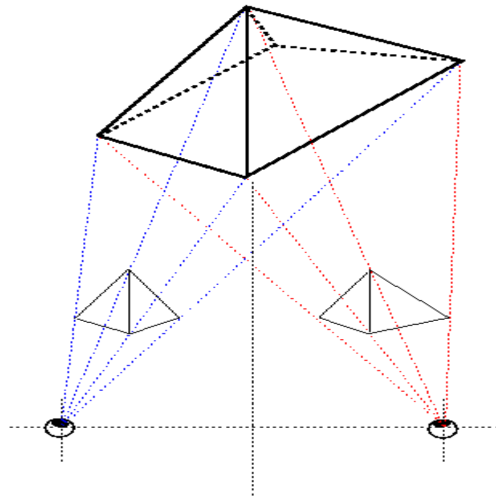


Figure 3. Principle of stereovision

✍ For stereovision, the graphics should present each of the two small pyramids to the corresponding eye.

2.1.2 Principle of Computer environment

Stereopsis is recreated in a computer environment by delivering separate left-eye and right-eye views in sequence. This can be done on a standard computer display if the user is wearing the appropriate shutter glasses (Figure 4). Crystal eyes, the glasses from Stereographics [1] that are used for this project, fall into this category:

Figure 4. Crystal Eyes glasses and emitter

These glasses employ liquid crystal in the lenses that alternately darken and become clear to block the incorrect view while transmitting the correct one on the monitor. When the viewer looks at the screen through active shutter eyewear, each shutter is synchronized to occlude the unwanted image and transmit the wanted image. Thus, each eye sees only its appropriate perspective view: the left eye sees only the left view (image output from the left buffer), and the right eye sees only the right view (image output from the right buffer). This kind of display is called a field-sequential stereoscopic display.

2.1.2 Quad buffering scheme

With normal non-stereo OpenGL double buffering, the drawn is typically done to the "back buffer" and then the buffers are swapped in order to put what has been drawn to the back buffer onto the "front buffer", which represents the visible display [3].

As it is mentioned before, one stereoscopic graphics can be viewed if only both left image and right image, a stereo pair, are displayed simultaneously. Thus, two double-buffered memories are required for stereoscopic display. That is, the framebuffer must be divided into four buffers which include front left buffer, back left buffer, front right buffer, back right buffer. Each individual double-buffered memory is employed to store information for either the left-eye image or the right-eye image. This architecture of framebuffer organization is called quad buffering: two full-resolution stereo pairs are rendered into two sets of buffers (for the right and left eyes).

Quad-buffered memory organization for the framebuffer is somewhat similar to the double-buffered system, and this can be visualised on Figure 5.

Figure 5. Double-buffered (left picture) and Quad-buffered (right picture) memory organisation

As one stereo pair is being displayed, a second pair is being rendered into the framebuffer. It means that both the left and the right back buffers are drawn, and then a single swap is done to put back stereo buffers content to the front (visible) stereo buffers.

This scheme produces the highest resolution and highest field-rate stereo images.
To display high-resolution stereo, the capacity of framebuffer for stereoscopic display needs to be twice as large as a monoscopic system. As an example, it can be seen on the picture above that if a 8MB framebuffer is sufficient to display a common real-time graphics image (with a certain resolution), then under the same configuration 16 MB (twice as much) is needed for stereographic.

2.1.3 Stereoscopic synchronization control

Two key synchronic controls guarantee a correct stereoscopic display:

✍ Hardware synchronization control.

Left buffer and right buffer switching must synchronize with vertical retrace of CRT.

✍ Software synchronization control.

Simultaneously switch two pairs of double buffers after finishing rendering each image.

2.1.4 Parallax

A stereoscopic display differs from a planar display by incorporating parallax.

When an electro-stereoscopic monitor image is observed without the eyewear, it looks like there are two images overplayed and superimposed. The refresh rate is so high that the viewer can't see any flicker, and it looks like the images are double-exposed. The distance between left and right corresponding image points is called parallax, and may be measured in inches or millimetres. Parallax and disparity are similar entities. Parallax is measured at the display screen, and disparity is measured at the retina. When wearing eyewear, parallax becomes retinal disparity. It is parallax that produces retinal disparity, and disparity in turn produces stereopsis. Parallax may also be given in terms of angular measure, which relates it to disparity by taking into account the viewer's distance from the display screen.

Parallax is classified into positive parallax, which produces an image distant from the viewer (behind the surface of the screen, Figure 6A), and negative parallax, which produces off-screen effects (images in viewer space, Figure 6B).

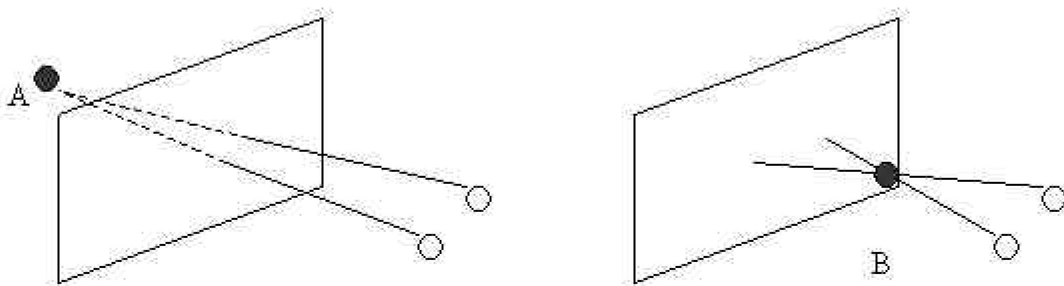


Figure 6. A: Positive parallax (left picture), B: Negative Parallax (right picture)

2.1.5 Introduction to Stereoscopic Perspective Projections

In order to put the different perspectives into the left-eye and right-eye buffers, it is necessary to generate projections that result in a stereoscopic effect that is both

geometrically correct and pleasing to look at [2]. As explained earlier, an element situated at positive or negative parallax appears ever in front or behind the display surface (Figure 7). An object appearing at the display surface is said to be at “Zero Parallax”.

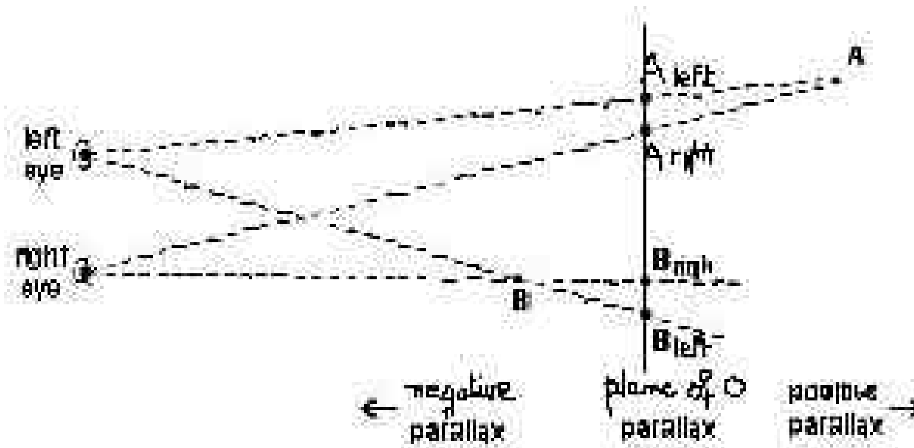


Figure 7. Perception of Parallax Effects

A good quality stereo image is composed of two stereo pair elements, each of which being a perspective projection whose camera (or “centre of projection”) is offset literally relative to the other camera position. The method is now explained in detail.

Let’s assume that there is a camera lying on the positive z-axis at (0,0,d) (Figure 8), d being the distance from the camera to the plane. Then (x, y, z) projects onto the plane at:

$\{xd/(d-z), yd/(d-z)\}$

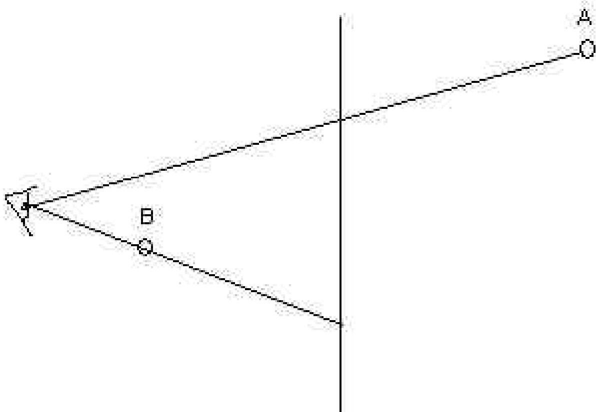


Figure 8. Assuming a single camera

Now let's introduce a camera offset to the projection.

To do a left-camera perspective projection, the camera is offset to the left by half the overall camera separation, c/2. To make the math easier, **the entire scene** is offset to the right instead of offsetting the camera to the left. So the left-camera projection of (x, y, z) now calculates to:

$$\{(x+c/2)d/(d-z), yd/(d-z)\}$$

For the right-camera projection the entire scene is offset to the left, so that the right-camera projection of (x, y, z) now calculates to:

$$\{(x-c/2)d/(d-z), yd/(d-z)\}$$

Generally, a visually pleasing, well-balanced stereo image makes use of both negative parallax and positive parallax, and at least some of the 3D scene projects at or close to zero parallax. Unfortunately, if the equations given above are used to calculate the stereo pair projections ALL possible points would project to negative parallax. It would thereofore look as if both A and B are between the viewer and the screen, which is not the case, as can be seen on Figure 9. All negative parallax tends to be uncomfortable to view.

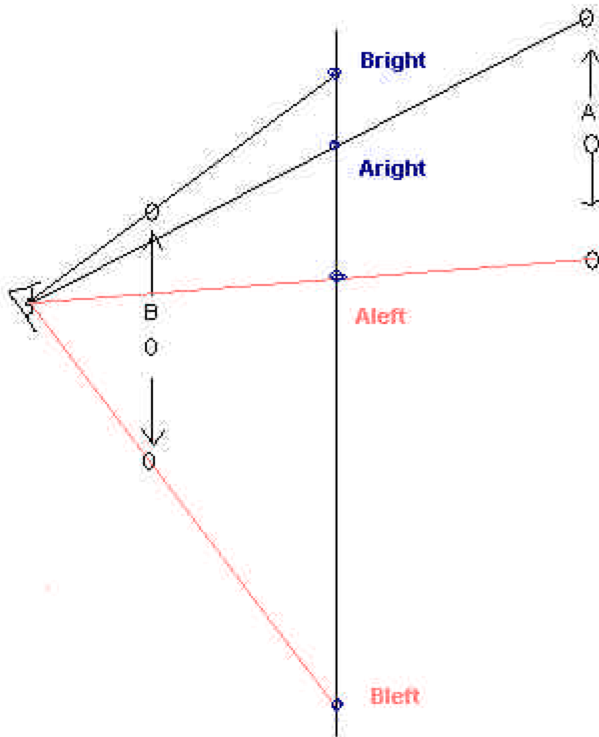


Figure 9. Offset applied to the whole scene --> ONLY negative parallax

It is possible to rectify by **shifting the projected values** leftward for the left-camera projection, and rightward for the right-camera projection. If the projected points are shifted by the same amount of the original camera offset, the resulting geometry places the original projection plane precisely to zero parallax. Scene elements originally placed in front of the projection plane projects to negative parallax, and scene elements originally placed behind the projection plane projects to positive parallax. The new equations for the projection are called "parallel axis asymmetric frustum perspective projections", and are for the left eye:

$$\{(x+c/2)d/(d-z)-c/2,\ yd/(d-z)\}$$

And for the right eye:

$$\{(x-c/2)d/(d-z)+c/2,\ yd/(d-z)\}$$

Figure 10 below illustrates the results given by these new equations.

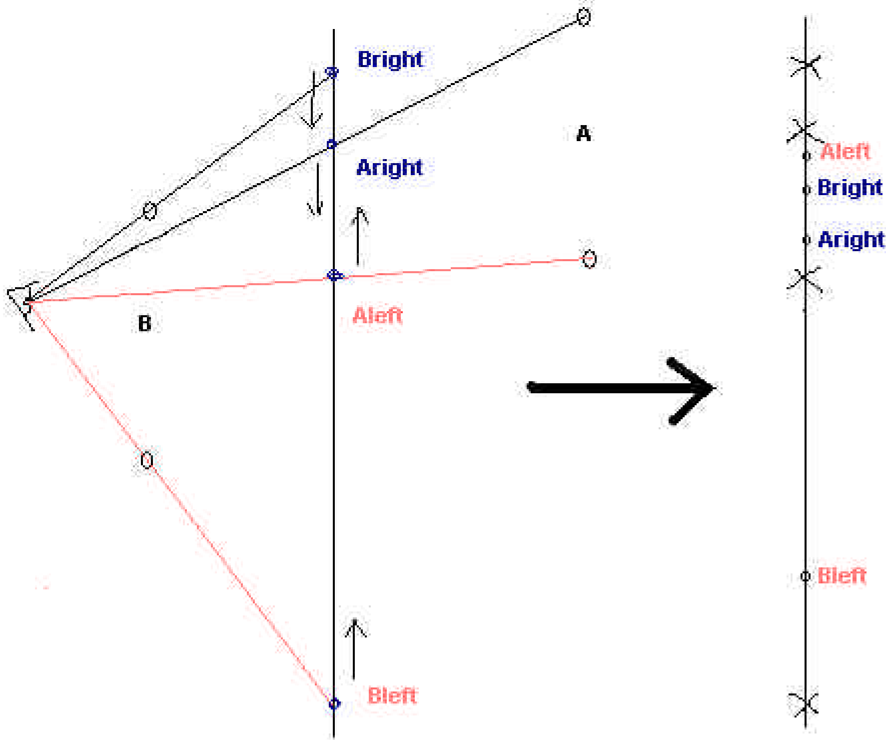


Figure 10. Shift of the projection --> negative AND positive parallax

Asymmetrical-frustum perspective-projections may be done in OpenGL using the standard OpenGL functions *glFrustrum ()* and *glTranslate ()*.

2.2 Software implementation for the stereo

2.2.1 Initialising a Window to enable stereo and testing it

In order to utilize OpenGL's stereo buffering support, a graphic window must be initialised using the WINDOWS API call *SetPixelFormat ()* with the pixel format descriptor flag PFD_STEREO set within a window's very first *SetPixelFormat ()* call (Indeed, Microsoft Windows only allow *SetPixelFormat ()* to be called once for any given window -- > subsequent calls for any given window are ignored) [1].

Here is some code called from the CMainWindow's class for enabling stereo and checking if it is successful or not. The program displays a message box in the case where stereo cannot be supported by the current configuration (this may be due to a lack of memory).

```
PIXELFORMATDESCRIPTOR pfd;            // pfd tells Windows how things are wanedt to be
// Iinitialiation of the pfd values
memset (&pfd, 0, sizeof (PIXELFORMATDESCRIPTOR)); // zero out all fields
pfd.nSize = sizeof (PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_GDI | PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER | PFD_STEREO;

// we may do test while doing this
 int iPixelFormat = ChoosePixelFormat(hdc, &pfd); // Windows find a matching pixel format
BOOL bSuccess = SetPixelFormat(hdc, iPixelFormat, &pfd); // The program sets the pixel format.
If the user's graphic configuration is not set up for stereo buffering, Windows will not allow the
PFD_STEREO flag. Therefore the code checks to see if the attempted stereo enabling actually succeeded in
changed the flag. This is performed in the most secure way.

iPixelFormat = GetPixelFormat (hdc);
DescribePixelFormat (hdc, iPixelFormat, sizeof (PIXELFORMATDESCRIPTOR), &pfd);
int RetVal;
//allows error checking while initialising the application.
if ((pfd.dwFlags & PFD_STEREO) == 0)
{// stereo mode is not accepted, a message box is displayed to prevent the user
        AfxMessageBox (_T ("The current display configuration does not support OpenGL stereo"));
        RetVal = FALSE;
}
else
{// yes, the  stereo mode is on
SetTimer (1, 20, NULL);
        RetVal = TRUE;
}
```

The error checking is then done in the initialisation function (This function is called *InitInstance ()* in the case of this project human model application), and the code corresponding to this inspection is the following one.

```
if (!((CMainWindow *)m_pMainWnd)->EnableStereoWindow())
{
        m_pMainWnd->ShowWindow (SW_HIDE);
        m_pMainWnd->DestroyWindow();
        exit(0);
}
```

Since the dialog box is being displayed in the Enable stereo function, this piece of code actually also checks for every further error checked in the Enable stereo function. If an error is found (e.g. if the stereo mode is not supported) the window is destroyed and the program quits.

2.2.2 Writing to Separate Buffers

The *glDrawBuffer ()* OpenGL function allows specifying which buffer subsequent OpenGL drawings and renderings should be directed to. Using GL_BACK_LEFT or GL_BACK_RIGHT (or even GL_BACK to act on both the back buffers) specifies the drawing buffer. To swap the buffers and call the back buffers to the front stereo buffers, a single call to *SwapBuffers ()* is enough. Below is a part of the code from the *OnPaint()* function, illustrating the stereo case.

```
glFlush();          // glFlush empties the buffers, causing all issued commands to be executed
                    // as quickly as they are accepted by the rendering engine
// Write in both the back buffers
glDrawBuffer (GL_BACK);
glClearColor(0.2f, 0.2f, 0.2f, 0.0f);          // non-black background, less ghosting
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    // Clears screen and depth buffer

// Draw the left eye vie
glFlush();
glDrawBuffer (GL_BACK_LEFT);
glPushMatrix();

// The asymmetric frustum projection is done first
```

```
StereoProjection(-6.4, 6.4, -4.8, 4.8, NEARBORDER,  FARBORDER, // X,Y,Z coord range
        1.75, 14.5, // Z-coord of zero-parallax "target", camera-target dist
        dfStereoMagnitudeAdjustment, dfParallaxBalanceAdjustment,
        LEFT_EYE_PROJECTION);

glRotatef(fRotAngle, 0.0f, 1.0f, 0.0f);
glTranslatef(0.0,Human.base_move,0.0) ;
Human.Draw_Model();
glPopMatrix();
// If a solid rendering is done, it would be wanted to clear the
// -buffer before each eye's rendering, since some stereo
// hardware/driver implementations only include one shared
// z-buffer
glClear (GL_DEPTH_BUFFER_BIT);

// Draw the right eye view
// The only difference with the left eye part is the last argument of "StereoProjection()"
glFlush();
glDrawBuffer (GL_BACK_RIGHT);
glPushMatrix();
StereoProjection(-6.4, 6.4, -4.8, 4.8, NEARBORDER,  FARBORDER, // X,Y,Z coord range
        1.75, 14.5, //Z-coord of zero-parallax "target", camera-target dist
        dfStereoMagnitudeAdjustment, dfParallaxBalanceAdjustment,
        RIGHT_EYE_PROJECTION);
glRotatef(fRotAngle, 0.0f, 1.0f, 0.0f);
glTranslatef(0.0,Human.base_move,0.0) ;
Human.Draw_Model();
glPopMatrix();
// Call SwapBuffers to put what is just drawn to backbuffers onto the actual display (front buffers)
glFlush();
BOOL bSuccess = SwapBuffers (hdc);
```

2.3 Motion control

Most animated characters are constructed using a skeleton. A skeleton is a connected set of segments, corresponding to the limbs, and the joints, a joint being a point where the

limb that is linked to the point may move. The limbs of the human model drawn in this application are represented by rectangles, but the model can still be represented as a skeleton. When the animator specifies the animation sequence, the motion is defined using this skeleton. The angle between the two segments is called the joint angle and, depending on the joint, it is of different nature. For example the wrist can be rotated in every direction, while the knee can only be bent in a single direction.

Three main methods to animate a skeleton exist: Geometric, Physical and Behavioural methods [19] [22][24].

2.3.1 Key framing

Skeleton animation consists of animating joint angles.

The key framing technique firstly identifies the number of key frames, frames where something important for the animation happens. At these key frames, the programmer explicitly calculates and passes the angles of every part of the model to the program via the animation function. Then the function uses these key frames to calculate the angles of each part of the body for every frame of the animation. Taking the angle between two key frames and dividing this angle among other frames accomplish this. For example, as shown in Figure 11, if the upper part of the leg has to be moved four degrees between two key frames and this has to be done in four frames, the function calculates that the upper part of the legs has to be moved one degree every single frame (four degrees divided by four frames equals one degree per frame), in order to accomplish the required motion.

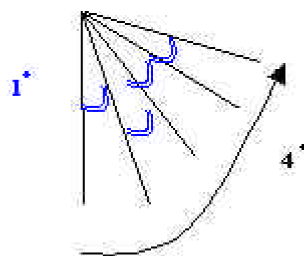


Figure 11. Key frames and single frames

2.3.2 Geometric motion control

In this kind of method, the motion is defined in terms of coordinates, angles and other shape characteristics. It may also be specified using velocities and accelerations, but no force is involved.

Using direct deformation of the model is a first kind of geometric motion control but it is very limited. This kind of motion is only useful for pre-calculated movements and it is not that realistic (since no force or acceleration is taken into account), but it may form the basis of more alternative techniques.

Another sort of geometric motion control is called inverse kinematics [25]. The forward kinematics problem consists in finding the position of end points (e.g. hand, foot) with respect to a fixed reference coordinate system as a function of time and without regard to forces or moments that causes the motion. The use of inverse kinematics permits direct specification of end positions. Joint angles are automatically determined based on the displacement of the skeleton. This is the key problem, because independent variables in a synthetic actor are joint angles. Unfortunately, the transformation of position from Cartesian to joint coordinates generally does not have a closed form solution. Typically, the inverse kinematics allows calculating the movement of the arm (the articulation angles) necessary to move the hand according to a certain trajectory. It is a very well known technique, but once again, it is used for pre-calculated animations. Inverse kinematics is usually used in coordination with key framing. Key frame systems are typical of systems that manipulate angles; for example, to bend an arm, it is necessary to enter the elbow angle at different selected times into the computer. Then the software is able to find any angle at any time using different kind of interpolation.

A higher level of specification of kinematics motion can be reached using constraints: the animator imposes a limb end to stay at a specified location or to follow a predefined trajectory. In the case constraints are added to a model, it is necessary to specify the priority of each constraint in the case they could not be simultaneously satisfied.

2.3.3 Physical motion control

Kinematics-based systems are generally intuitive and lack dynamic integrity. The animation does not seem to respond to basic physical facts like gravity or inertia. Only modelling human that moves under the influence of forces and torques can be realistic.

Forces and torques cause linear and angular accelerations. The motion is obtained by the dynamic equation of motion. These equations are established using the forces, the torques, the constraints and the mass properties of objects. This work deals not only with describing movement, but also simulating a human model with gravity and forces included. It allows more elaborated movements, but it is still a situation of databank, except that they are not fixed anymore but adapted to the situation. (A good example would be the realization and the animation of hairs. Great things have been done in this area, simulating hairs using cylinders and their interactions, as it has been done in the recent movie “Final Fantasy: The Spirits Within” based on the popular video game of the same name (http://www.sgi.com)).

2.3.4 Behavioural motion control

It is however possible to reach a higher degree of description using task planning. Instead of describing the movement themselves, it is possible to depict the actions to be done. That’s based on the relation existing between the objects and their relation with the human model itself.

Task modelling may be divided into three phases:

✍ World modelling: it consists of describing the geometry and the physical characteristics of the objects (e.g. golf club,) and the human model.

✍ Task specification: a task specification by a sequence of model states using a set of spatial relationships or a natural language interface is the most suitable and popular.

✍ Code generation: several kinds of output code are possible: series of frames ready to be recorded, value of parameters for certain key frames, script in an animation language or a command-driven animation system.

In each case, the correspondence between the task specification and the motion is very complex.

Behavioural animation corresponds to modelling the behaviour of characters, from path planning to complex emotional interactions between characters. In an ideal implementation of a behavioural animation, it is almost impossible to exactly replay the same scene twice. For example, in the task of walking, everybody walks more or less the same way, following more or less the same laws. This is the “more or less” which is difficult to model. And also a person does not walk the same way every day. If the person is tired, or happy, or just got some good news, the way of walking is slightly different.
Anyway in the case of this work, which is intended to train golf player, this might be a much too complicated approach.

2.4 Motion capture

Getting angles of the animation (fixed or adapted to the situation) seems to be something very useful according to the different type of motion control seen, and it’s an interesting way of dealing with getting a perfect databank for the movement we want the person to train to do in a perfect way. Asking a golfer to do a swing (as an example) in a perfect way would allow the desired data to be obtained, and then use them to train every subsequent player by comparing it with the captured player movement. Indeed, it would be possible for the user to visualise its own movement, and compare it to the perfect one that is recorded (this one being adapted to the size of the user). There are mainly two kinds of motion capture: the magnetic motion capture and the optical motion capture [23].

2.4.1 Magnetic motion capture

Magnetic motion capture systems use sensors to accurately measure the magnetic field created by a source. Such systems are real-time, in that they can provide from 15 to 120 samples per second (depending on the number of sensors) of 6D data (position and orientation) with minimal transport delay. A typical magnetic motion capture system has one or more electronic control units into which the source(s) and sensors are cabled. The electronic control units are, in turn, attached to a host computer through a network or serial port connection. The motion capture or animation software communicates with these devices via a driver program. The sensors are attached to the scene elements being

tracked. The source is set to be either above or on the side of the active area. There can be no metal in the active area, since it can interfere with the motion capture. The obvious way to animate character using magnetic motion capture is to place one sensor at each joint. However, the physical limitations of the human body (the arms must be connected to the shoulder, etc.) allow an exact solution with significantly fewer sensors.

Because a magnetic system provides both position and orientation data, it is possible to infer joint positions by knowing the limb lengths of the motion-capture object. With a magnetic system, and according to the application human model, the method to drive the model would be to drive joints directly by deriving angles from two sensors, placed evenly between joints of the actor. Because the magnetic system provides data in real time, the director and actors can observe the results of the motion capture both during the actual take and immediately after, with audio playback and unlimited ability to adjust the camera for a better view.

An advantage of this method is that it allows for fewer sampling locations and less inferred information, while allowing interactive display and verification of the capture data, providing closed loop models where talent, direction and production can all participate directly in the capture session. Another point of interest being its cost, which is typically under 40,000 US dollars, substantially less than optical full-body systems.

However, its sensitivity to metal is a critical issue. Care must be taken that the stage, walls, and props for a motion capture session are non-metallic. In terms of accuracy for entertainment production such as the one proposed in this application, it might be a real problem if the golf club is made with metal! Another point of concern with this method is that the sampling rate might be too low for many sports moves. For body tracking applications, magnetic systems tend to have 30 to 60 Hz effective sampling rates. A fastball pitcher's hand moves at roughly 40 meters per second, approximately a meter per sample. Also, filtering is typically used to compensate for measurement jitter, reducing the effective range to 0 to15 Hz. In the case of a golf player, movements may sometimes be too fast as well!

Apart from these problems, it might be noticed that the maximum effective range of these devices is substantially less than the maximum possible for optical systems, and that there is a limitation and encumbrance when attaching 10 to 20 fairly thick cables to a human subject.

2.4.2 Optical Motion Capture Systems

These systems are based on high contrast video imaging of retro-reflective markers, which are attached to the human whose motion is being recorded. The markers are small spheres covered in reflective material. High-speed digital cameras image the markers. The number of cameras used depends on the type of motion capture. Facial capture usually uses one camera, sometimes two. In this project, full motion capture would be done. It may use four to six (or more)) cameras to provide full coverage of the active area. To enhance contrast, each camera is equipped with infrared (IR) emitting LEDs and IRs (pass) filters are placed over the camera lens. The cameras are attached to controller's cards, typically in a PC chassis. Depending on the system, either high-contrast (1 bit) video or the marker image centroids are recorded on the PC host during motion capture. Before motion capture begins, a calibration frame (a carefully measured and constructed 3D array of markers) is recorded. This defines the frame of reference for the motion capture session.

After a motion capture session, the recorded motion data must be post-processed or tracked. The centroids of the marker images (either computed then, or recalled from disk) are matched in images from pairs of cameras, using a triangulation approach to compute the marker positions in 3D space. Each marker's position from frame to frame is then identified. Several problems can occur in the tracking process, including marker swapping, missing or noisy data, and false reflections.

Tracking can be an interactive and time-consuming process, depending on the quality of the captured data and the fidelity required. For straightforward data, tracking can tale anywhere from one to two minutes per captured second of data (at 120 Hz). For complicated or noisy data, or when the tracked data is expected to be used as is, tracking time can climb to 15 to 30 minutes per captured second, even with expert users. First time

users of tracking software can encounter even higher tracking times. Optical motion capture service providers charge as much as 180 dollars per hour for tracking.

For a human body, which is the subject of this application, typical set up for its animation involves 20 to 30 markers glued (preferably) to the subject's skin or to snug fitting clothing. Markers range from 1 to 5 cm in diameter, depending on the motion capture hardware and the size of the active area. Marker placement depends on the data desired. A single marker is attached at each point of interest, such as the hips, elbow, knees, feet, etc…

A simple configuration in accordance with the project human model would attach three markers to the subject's head on a hat or skull cap, one marker at the base of the neck and the base of the spine, a marker on each of the shoulders, elbow, wrist, hands, hips, knees, ankles and feet, 21 markers in total, as can be seen on the model below (Figure 12).

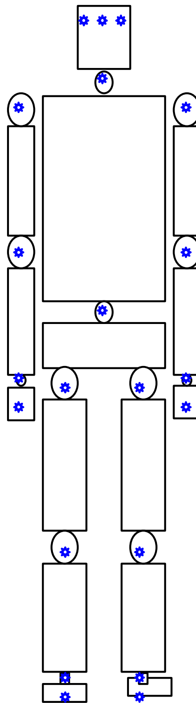


Figure 12. Motion capture simple marker configuration

A good benefit of this method is that depending on the system used and the precision required, the motion capture area can be arbitrarily large, and the fact that the subject is not attached to the motion capture system allows more freedom of movements. Also, additional markers cost very little: theoretically, hundreds of markers could be included in a given scene. However, given the problems of occlusion and the limitations of the tracking software, the practical maximum is probably less. A point of major interest compared to the magnetic motion capture is that the sampling rate is high enough for most sports moves. At a 120 to 250 Hz sampling rate, most human motions are easily measured. However, two classes of motions actually are on fringes of this sample: pitching and impact. And while doing a swing, the golfer hits the ball via its club. When throwing a 90 m.p.h (miles per hour) fastball, the human hand travels 33 cm in 1/120 second. For impact events such as drumming, hitting, and hard falling, accelerations may have frequency components beyond 120 Hz. While executing a swing, a golf players arms might be very fast as well. Thankfully these motions are a blur for human observers and the loss of accuracy is usually imperceptible.

Among the disadvantages of this method, the cost can be seen as a major limitation, since at 150,000 to 250,000 US Dollars, optical systems are the most expensive motion capture systems described. The operational costs are also higher, being more similar to film or video production, while the tracking time can be much greater than the actual capture session and may vary unpredictably, depending on accuracy requirements, motion difficulty, and the quality of the row data captured. Another point of concern is the sensitivity to light of these systems: since current optical systems are contrast based, backgrounds, clothing, and ambient illumination may all be issues. The sensitivity to the reflection implies that wet or shiny surfaces (mirror, floors, jewellery, and so on) can cause false marker readings, requiring to play golf (since it is what we re concerned with) in very specific conditions. Moreover, since a marker must be seen by at least two cameras (for 3D data), the total or partial occlusion caused by the subject, props, floor mats or other markers, can result in lost, noisy, displaced, or swapped markers. Common occlusions are hand versus hip (standing), elbow versus hip, or hand versus pop in hand or opposite hand, and these kinds of situations may obviously happen in the case of a golf

player! In addition to all this, since it is non-real-time device, unlike magnetic motion capture, there is no immediate feedback regarding the quality and character of captured data, it is impossible to know if a given motion has been adequately captured. Because of this, two to three acceptable takes must be completed to ensure a probability of success if additional capture sessions are not feasible to acquire missed data. And while additional markers are relatively cheap, more measured points are required, as the joints angles must be inferred by the rays connecting the tracking process, removing the position-only restriction from optical data. However this does add complexity to the tracking process. Finally, since multiple cameras are used, the frame of reference for each camera must be accurately measured. If a camera is misaligned (due to partial marker occlusion, or a simple tripod bump), markers measured by that camera are placed inconsistently in three space relative to markers measured by other cameras. This is particularly troubling at hand-off (the time a marker passes from one camera's field of view to another's), as duplicate points may be created from the same marker or the marker path may jump.

Dealing with motion capture may be useful depending on how this project is going to be continued other the three coming years. This is discussed in chapter 6.

The theoretical and technical background necessary before addressing the problem of human modelling in 3-D is performed in this chapter. The sense of three-dimensionality, which results from how the brain processes what the eyes see, can be recreated in a computer environment by delivering separate left-eye and right-eye views in sequence (the corresponding software implementation is therefore explained). Motion control is then discussed (three main methods to animate a skeleton exist), including an explanation of the concept of key framing. In the following chapter an explanation of the first steps performed during this project is done, allowing the reader to become familiar with the OpenGL library.

Chapter 3

3. Design and Implementation

This chapter explains how to start a human modelling project as well as the necessary basics for a beginner. Indeed, learning how to draw basic objects such as cubes and pyramids is a logical start before the drawing of a human model. The following step being to rotate and move these forms in 3D, and finally add some stereo effects to them. A working version of this simple model is available at the moment, so that people can build on it and do their "own first steps". To deal with three-dimensionality, it is necessary to be familiar with 3-D graphics programming techniques such as OpenGL[6-10] (for "Open Graphics Library"), C++[12-15] and Stereographics API [1-2]. Therefore, without going into too much detail, some very useful functions of the OpenGL programming language are described. It is very basic, but it allows a better understanding of the essentials of the application code, as well as how OpenGL allows a programmer to produce high quality graphical colour images of three-dimensional objects. Moreover, the main steps to create a new project are described in this part, with thought for the Ph.D student who is going to continue this project over three years.

3.1 First steps

3.1.1 Building a new project

It is first of all indispensable to create a Win32 Application in Visual C++.

Then, it is essential to link the required OpenGL libraries. Therefore:

✍ Go to Project, Settings, and then click on the Link tab.
  Under "Object/Library Modules" at the beginning of the line, add
  OpenGL32.lib Glu32.lib GLaux.lib Glut32.lib

Since some problems were met while installing the glut library, it is explained in detail in the Appendix 1.

It is now possible to write an OpenGL Windows program.

Since some stereo effects are required for the application, one more change in the project settings is indispensable.

✍ Go to Project, Settings again, and click C/C++

Under “Category”, choose “Code Generation”.

Then under “Use run-time library”, choose “Multithreaded DLL”.

It is now possible to start programming without any worries.

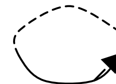
3.1.2 Work done in the 3-D World

The first things to learn are how to set up an OpenGL window, how to draw 2-D objects, and finally learn how to do all this in a 3-D world. How to make objects spin around one or more axis, and how to add colour and light, or how to use texture mapping in order to make them look better (thanks to Jeff Molofee’s Tutorials [7]) is also something necessary to learn.

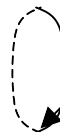
The result of this can be seen on

Figure 13. It contains a cube and a pyramid rotating around two different axes in OpenGL.

The Pyramid is rotated on the Y-axis, from left to right



The Quad is rotated on the X-axis, from back to front



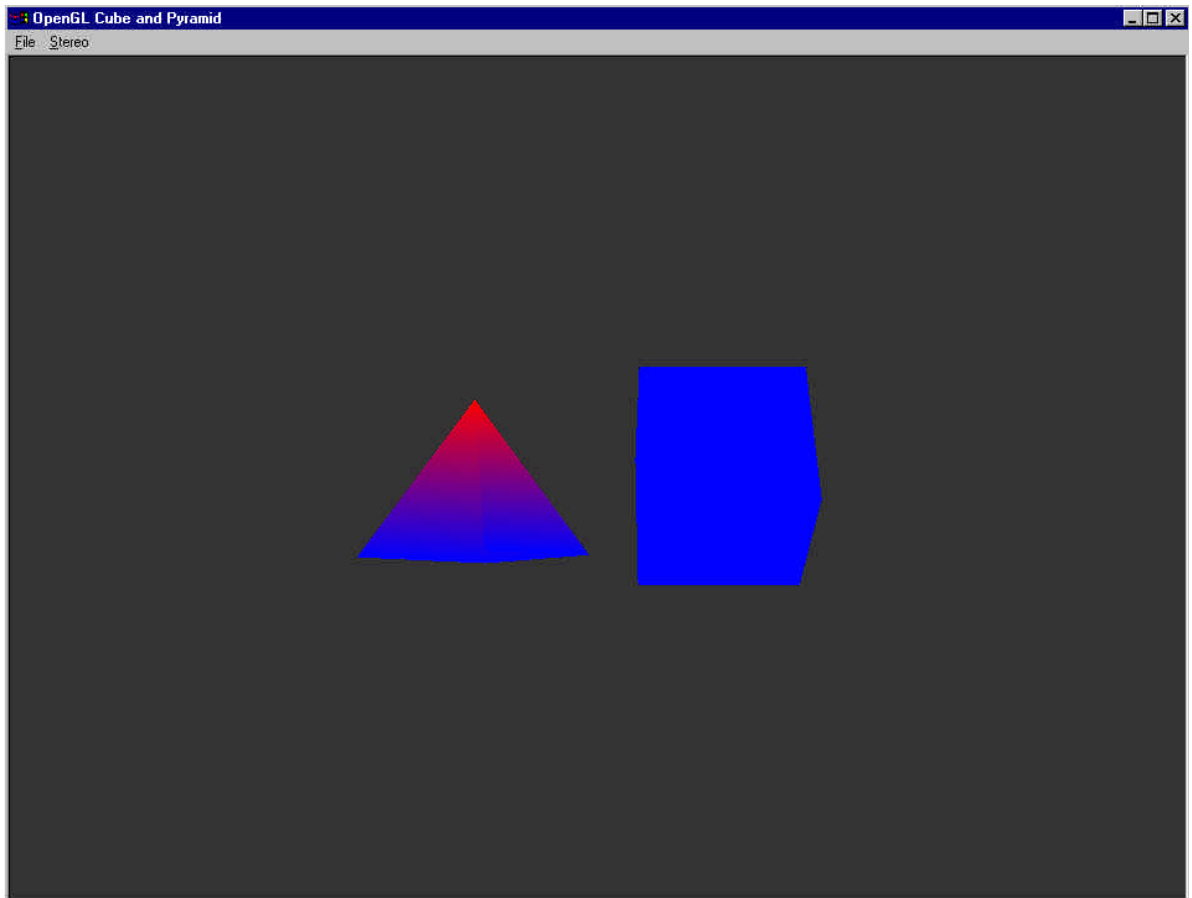


Figure 13. First Application

At this stage, no stereo effects are available; it is just3-D ON the monitor, as on a normal sheet of paper. So adding stereo effects appears to be the next step.

3.1.3 Stereo Effects

Creating stereoscopic images on this application allows adding effects with the 3-D graphics. What mainly needs to be done in order to add stereo effects is described below.

Firstly, it is necessary to perform the asymmetric frustum described in Chapter 2. A function therefore needs to be built. It is called *StereoProjection ()* and it is a reusable function.

The only difference then with the drawing done without stereo is the use of two front (and two back) buffers instead of a single one, as explained either in the previous chapter.

(Of course a call to the StereoProjection function is required just before the effective drawing). It is done in the *OnPaint ()* function. The resulting stereo aspect works very well with the glasses, and is comfortable to look at. In the actual application, a choice is offered to the user who can decide to view the scene either using stereo effects or not. This choice is made available via a menu. In the case where the viewer wants no stereo, the drawing is just done in a single buffer and a single non-stereo projection is consequently generated.

The resulting program is a working version. Therefore, further people working on this project can use this **executable** to draw their own model: they have the opportunity to build on it and learn by their own.

3.2 OpenGL basics

This part might especially interest a further programmer, since the meaning of some of the main functions is given.

3.2.1 Drawing basics

It may be useful to any further programmer to know that every OpenGL program is linked to a rendering context (RC), and that it is necessary to create a Device Context (DC) to draw a window. The DC connects the Window to the Graphic Device Interface, and the RC connects OpenGL to the DC. Moreover, the meaning of some very useful functions to draw a model might be a real help as well.

✍ glLoadIdentity ()

It allows the user to move back to the centre of the screen. It is very functional to reset the view after a rotation. Indeed, after a rotation, very unexpected results may happen when translating and drawing another object! It's better to go back to "0" and move from here.

✍ glTranslate (x, y, z)

It allows the user to move a set amount from the current screen position.

✍ glBegin () and glEnd ()

These functions are used to start and end creating a geometric form such as a triangle or a cube. Actually, the glut library provides very handy functions to draw predefined forms, without having to define the x, y and z values of each of the corners.

✍ glColor3f (float, float, float)

This allows setting the colour of any object.

✍ glRotatef (Glfloat angle, GLfloat x, Glfloat y, Glfloat z)

It simply provides rotation. The parameter 'angle' specifies the angle of rotation in degrees. The parameters x, y and z are the coordinates of a vector, respectively. It's a good practice to make objects either clockwise or counter clockwise, and it's better not to mix the two unless there is a good reason.

There are two types of light. The ambient light, which doesn't come from any particular direction, and the diffuse light, which is created by the light source and provides nice shading effects.

3.2.2 Other useful functions

Thanks to the msdn library [11] for this part of the report.

✍ glFlush

It empties all the buffers (network buffers and the graphic accelerator itself) used causing all issued commands to be executed as quickly as the actual rendering engine accepts them. All programs should call this function whenever they have all of their previously issued commands completed. For example, this function is used in the *OnPaint ()* function of the executable application before the drawing in a new buffer starts.

```
 // Draw the left eye view
      glFlush();
      glDrawBuffer (GL_BACK_LEFT);
--------------Other code ----------------
```

```
// Draw the right eye view,
// The only difference with the left eye part is the last argument of "StereoProjection()"
        glFlush();
        glDrawBuffer (GL_BACK_RIGHT);
--------------Other code ----------------
```

✍ glPushMatrix & glPopMatrix

The current matrix in any mode is the matrix on the top of the stack for that mode. *glPushMatrix* pushes the current matrix stack down by one, duplicating the current matrix. That is, after a *glPushMatrix* call, the matrix on the top of the stack is identical to the one below it. *glPopMatrix* pops the current matrix stack, replacing the current matrix with the one below it on the stack. OpenGL provides the means to build hierarchical models through the functions *glPushMatrix* and *glPopMatrix*. As an example, in the function *Draw_base_Legs ()* of the human model application, these functions are used in order to retrieve the matrix prior to the translations.

```
void CBodySize::Draw_Base_Legs(void)
{
  glPushMatrix() ; // Save the current matrix before the  first  translation
  glTranslatef(0.0,-(BASE_HEIGHT/2+UP_LEG_JOINT_SIZE),0.0) ;
  glPushMatrix() ; // Save the current matrix after the translation to the left side
  glTranslatef(TORSO_WIDTH/2-UP_LEG_JOINT_SIZE,0.0,0.0) ;
  Draw_Leg(LEFT) ;
  glPopMatrix() ; // Restore the matrix before the translation to the left
  glTranslatef(-TORSO_WIDTH/2+UP_LEG_JOINT_SIZE,0.0,0.0) ;
  Draw_Leg(RIGHT) ;
  glPopMatrix() ; // Restore the first matrix saved: the one before any translation
}
```

✍ BEGIN_MESSAGE_MAP(theClass**,** baseClass),

This macro is used to begin the definition of a message map; 'theClass' specifies the name of the class whose message map this is. 'BaseClass' specifies the name of the base class of theClass. In the implementation file that defines the member functions for the

class (cube.cpp in the human model application case), this means to start the message map with the BEGIN_MESSAGE_MAP macro, then add macro entries for each of the message-handler functions, and finally complete the message map with the END_MESSAGE_MAP macro. A part of the human model code is commented below, in order to illustrate this a little bit better, and to explain some of the functions used in this message map.

```
// Defines the member functions of the class (associates messages with member functions)
BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
ON_WM_CREATE()
// The ON_WM_PAINT  macros (from the Microsoft Foundation Class Library) map common message to
// default handler function. It is called whenever Windows sends a WM_PAINT message
ON_WM_PAINT ()
ON_WM_SIZE () // idem
ON_WM_TIMER ()

// ON_COMMAND: indicates which function will handle a command message from a command user-
// interface  object such as a menu item or toolbar button.
// Arguments are  (command ID, name of the message-handler function to which the command is mapped)
// This macro is inserted manually or by the ClassWizard.
ON_COMMAND (IDM_FILE_ROTATION, OnFileRotation)

// There should be exactly one ON_UPDATE_COMMAND_UI macro statement in the message-map for
// every user-interface update command that must be mapped to a message handler.
// Arguments are the message ID and the name of the message-handler function to which the message is
// mapped.
ON_UPDATE_COMMAND_UI (IDM_FILE_ROTATION, OnFileRotationUpdate)

ON_COMMAND (IDM_FILE_ANIMATION , OnAnimate)
ON_UPDATE_COMMAND_UI (IDM_FILE_ANIMATION , OnAnimateUpdate)
------------------ And more entries to handle additional commands ----------------------------
--------

// Complete the message map
END_MESSAGE_MAP ()
```

✍ OnTimer

Map Entry	Function Prototype
ON_WM_TIMER( )	afx_msg void OnTimer(UINT nIDEvent);

The framework calls this member function after each interval specified in the *SetTimer* member function used to install a timer. Its use is illustrated by the following peace of code from the human model application.

```
void CMainWindow::OnTimer (UINT nIDEvent)
{
        if (bRotationOn)   // Change y-axis rotation angle
        {
                fRotAngle += fRotationSpeed;
                if (fRotAngle >= 360.0f)
                        fRotAngle -= 360.0f;
                glFlush();
                Invalidate();        // Continuously redraw while rotating
        }
        if(bAnimationOn) // Very important: it allows to make the man walk continuously!
        {
                Human.animate_body(angles);
                glFlush();
                Invalidate(); // And redraw the image each time
        }
}
```

In this chapter the basics of OpenGL and Stereo effects are explained, as well as some useful functions. Indeed, it is necessary to be able to draw basic forms such as Cubes and Pyramids, without/with stereo effects, before starting to draw a complete human model. The next chapter investigates the creation of a walking human bottom model, by building

on the executable containing the first steps (quad and pyramid rotating around two different axis).

Chapter 4

4. The Frame Bottom Model

This chapter explains how to create and animate the bottom of a human model that is drawn using cubes and spheres. The lower part of the body is built, first drawing the base and the legs of the model in a static way, before animation is added. The walking animation of the bottom is not read from a file but instead exists in the body of the animation function, and is based on the technique of key framing. In this section, a function is provided for the user that draws a basic model of the bottom of a human body. There is also a function that is able to calculate the vertical displacement of this particular model, and another function that deals with the animation of the model.

4.1 Building the basic static model

The bottom model is based on the model found on dev-gallery site [17]. It is composed of 6 joints, a base, two upper legs, two lower legs and two feet (Figure 14). A joint is the point where the limb that is linked to the point may move.

.

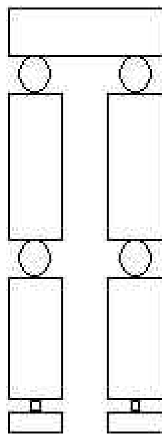


Figure 14. Bottom model components

This bottom model is symmetrical, so in order to avoid repetition, it is necessary to build it in a hierarchical fashion. Acting methodically also allows each part of the model to move in accordance with the whole body without any exterior help. OpenGL provides the means to build hierarchical models through the functions *glPushMatrix ()* and *glPopMatrix ()*. Therefore, the top most part in the hierarchy is the base of the body, followed by the upper leg joint, the upper leg, the lower leg joint, the lower leg, the foot joint and then the foot (Figure 15). In this way, if a rotation is applied to the base, all the other parts rotate as well, whereas if a rotation is applied to the lower leg joint, only the parts lower from it rotate. The model at this point is constructed of three main parts: the base, and the two legs.

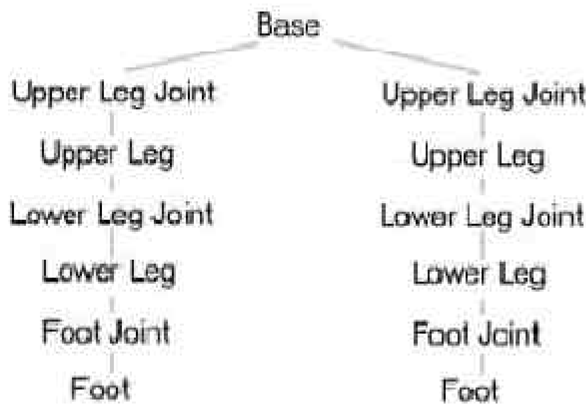


Figure 15. Bottom model hierarchy

In order to construct this model, some relative metrics were used to approximate the human body and are declared as #define values. Only the TORSO_HEIGHT has a defined size, and all the other parts of the bottom are related to this height.

```
#define FOOT_JOINT_SIZE        TORSO_HEIGHT * 0.0625
#define FOOT_HEIGHT            FOOT_JOINT_SIZE * 2.0
#define FOOT_WIDTH             LO_LEG_WIDTH
```

```
#define FOOT                    FOOT_WIDTH * 2.0
#define TORSO_HEIGHT            0.8
#define BASE_WIDTH              TORSO_HEIGHT * 0.75
#define BASE_HEIGHT             TORSO_HEIGHT / 4.0
#define UP_LEG_HEIGHT           TORSO_HEIGHT * 0.5
#define UP_LEG_JOINT_SIZE       TORSO_HEIGHT * 0.125
#define UP_LEG_WIDTH            UP_LEG_JOINT_SIZE * 2.0
#define LO_LEG_HEIGHT           UP_LEG_HEIGHT
#define LO_LEG_WIDTH            UP_LEG_WIDTH
#define LO_LEG_JOINT_SIZE       UP_LEG_JOINT_SIZE
#define LEG_HEIGHT              UP_LEG_HEIGHT + LO_LEG_HEIGHT + FOOT_HEIGHT +
                    2*(FOOT_JOINT_SIZE+UP_LEG_JOINT_SIZE+LO_LEG_JOINT_SIZE)
```

At this stage, a simple change of the torso height in the code allows to change the whole size of the bottom in a "nice looking" way. Anyway, the sizing of the model is done using a more intelligent method in Chapter 5, while drawing the whole body. Indeed, a class is made for the model allowing its size as well as its name to be changed dynamically.

4.2 Animation and implementation of the model

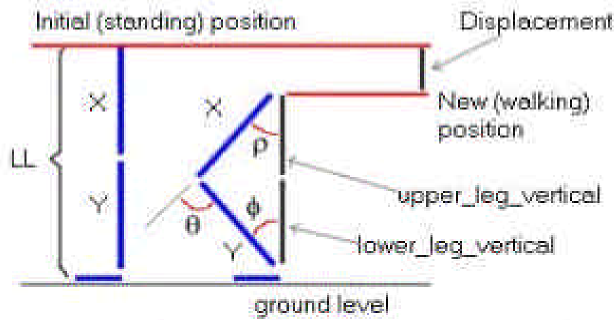
The animation of this bottom model is then done using the concept of key framing. A walking animation function called *animate_base ()* is built on this purpose.

4.2.1 Vertical displacement of the body

Due to the walking animation, there is a vertical displacement of the body, and the function that calculates this displacement is the function *find_base_move ()*. At this point, the vertical displacement due to the foot is not taken into account. The vertical displacement (VD) can be calculated by subtracting the values *upper_leg_vertical* and *lower_leg_vertical* by the leg's length (LL):

$$VD = LL - (upper_leg_vertical + lower_leg_vertical)$$

The Figure 16 below illustrates this.



Displacement =

Displacement =

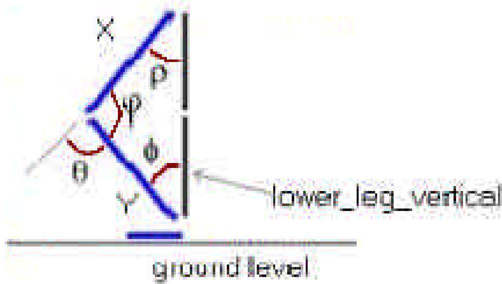
LL – (upper_leg_vertical + lower_leg_vertical)

upper_leg_vertical = X * cos(?)

lower_leg_vertical = Y * cos (?)

Figure 16. Vertical Displacement

Basic geometry is used to find ? (Figure 17), since the angle available and which is the lower leg angle is actually ?.



?+?+? = 180? and ?+? = 180?

? = ? - ?

Figure 17. Angle for the lower vertical leg

The angles are first converted from degrees to radians (as the library routine *cos* that is used to find the cosine angles needs the angles to be in radians).

The vertical displacement for both legs is found and then checked to see which one of the two is touching the ground (its vertical displacement is less than the others is); this value is then returned by the function. Below is the code corresponding to this function.

```
//displacement due to the foot is taken into account
double CBodySize::find_base_move(double langle_up, double langle_lo, double rangle_up, double
rangle_lo)
{
    double result1, result2, first_result, second_result, radians_up, radians_lo ;

    radians_up = (PI*langle_up)/180.0 ;     // Converts angle from degree to radian to be used by "cos"
    radians_lo = (PI*langle_lo-radians_up)/180.0 ;
    result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
    result2 = (LO_LEG_HEIGHT + 2*(LO_LEG_JOINT_SIZE+FOOT_JOINT_SIZE)+FOOT_HEIGHT)
            * cos(radians_lo) ;
    first_result = LEG_HEIGHT - (result1 + result2) ;          // For the left leg

    radians_up = (PI*rangle_up)/180.0 ;
    radians_lo = (PI*rangle_lo-radians_up)/180.0 ;
    result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
    result2 = (LO_LEG_HEIGHT + 2*(LO_LEG_JOINT_SIZE+FOOT_JOINT_SIZE)+FOOT_HEIGHT)
            * cos(radians_lo) ;
    second_result = LEG_HEIGHT - (result1 + result2) ;        // For the right leg

    if (first_result <= second_result)         // Check which one is touching the ground
    return (- first_result) ;                  // --> Its vertical displacement is less than the other
    else
    return (- second_result) ;
}
```

4.2.2 Walking animation

The animation function is a walking animation cycle based on eight key frames, as described on the book by Tony Wight "Moving Pictures" [17]. The first four key frames are used for the purpose of animating the first half of the walking movement, and the rest four in order to animate the second half. In the first part of the animation, the leg that is in front before the animation starts ends up behind and the leg that is behind ends up in front. In this particular case, the second half of the animation does the reverse of the first half of the animation in order to complete the walking cycle, and starts from the beginning for a new cycle.

An interesting feature of this model is the possibility for the user to dynamically increase or decrease the number of frames between two key frames. This is done via the menu, in steps of 2 in order to preserve the symmetry of the animation model used. It allows the user to increase the degree of precision (giving the feeling that the model is moving more slowly), or on the contrary to decrease this degree of precision by using fewer frames between two key frames.

After compiling and running the program, the user can see the bottom model walk, as shown on the screen shots taken from this program in Figure 18.

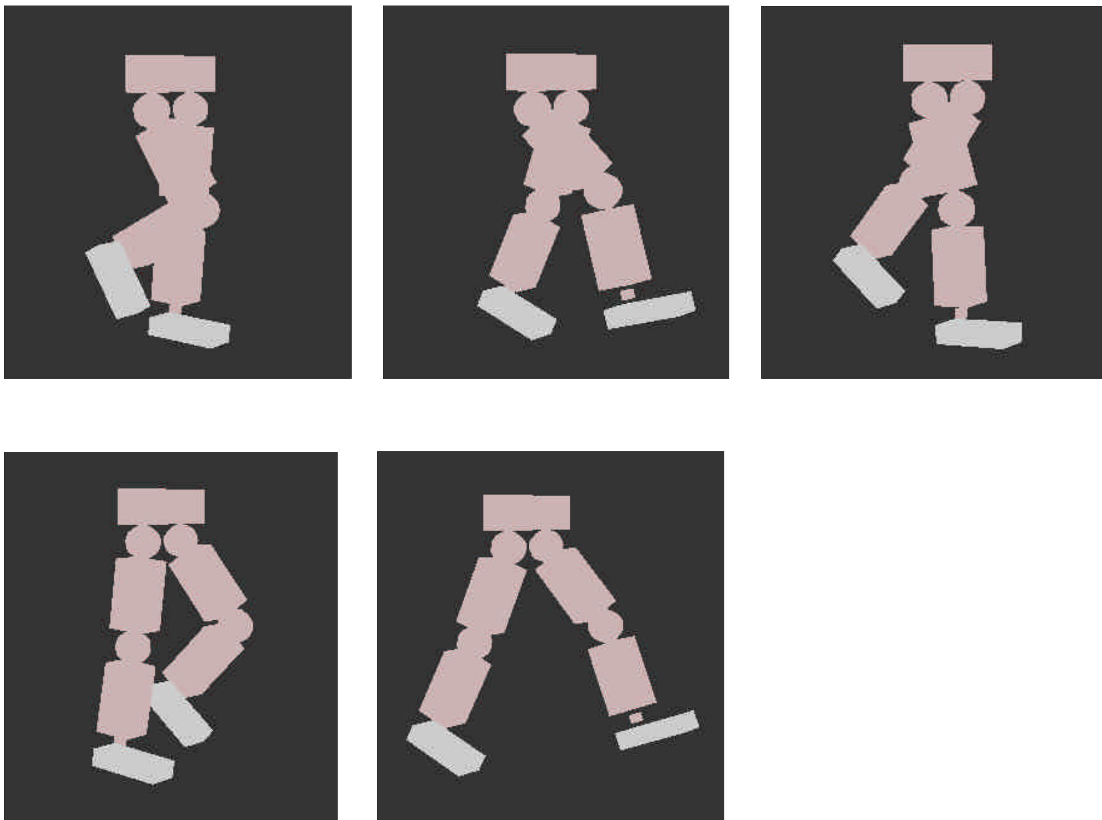


Figure 18. The "Animated" Bottom Model

The animation code is explained in detail in chapter 5, in the more general case where the angles are read from a file.

4.3 Class behaviours for the application code

The classes of the applications are briefly described in this section, so that it should allow an easier understanding of the code.

✍ CMyApp:

Just creating this application object runs the whole application.

CMyApp myApp;

✍ InitInstance:

When any *CMyApp* object is created, this member function is automatically called. Any data may be set up at this point. Also, the main window of the application should be created and shown here. This function returns TRUE if the initialisation is successful. *CMainWindow* message map associates messages with member functions (defines the member functions of the class).

At this stage, the walking animation exists in the body of the animation function, and the model is based on the technique of key framing. The precision can be changed dynamically via the number of frames between two key frames. It is also possible to zoom in or out in the case the stereo is not on. The classes' behaviours are explained as well. The creation of the whole walking model is explained in Chapter 5.

Chapter 5

5. The Complete Human Model

The bottom model of the man is completed in this chapter to draw the complete three-dimensional wire frame model of a man. This human model is still based on rectangles and spheres, and it also has a predefined walk. Anyway the animation function is different from that of the bottom model. The program is still using the key frame concept, but this time it reads the walking angles from a file, which is better than having a hard coded model. The model's size and name can dynamically be changed using a dialog box displayed from the menu. This is very functional according to the final purpose of this project, since it permits mapping the model back to the person being trained. The use of a 'CBodySize' class to represent the model (instead of using a particular drawing and defined values) is a better approach and facilitates this kind of functionality.

5.1 Building the static model

5.1.1 Design of the model

The entire model is composed of 14 joints, a base, two upper legs, two lower legs, two feet, a torso, a head, two upper arms, two lower arms and two hands (Figure 19). This model is inspired from the model built by Fotis Chatzinikos [17].

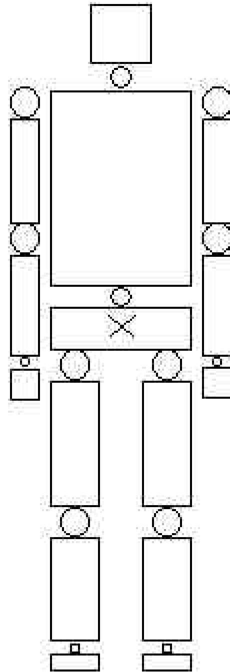


Figure 19. Human model components

A "CBodySize" class is used instead of *#define* values as in the bottom model. It allows access to the sizes and the creation of a real instance of the model (which means that the code may be modified to create more than one instance of a human). This class includes the size of the different parts of the model. The sizes are private. Indeed, it is more secure to access this data via a function rather than leaving them pubic. It also includes the drawing functions (which are not in the "CMainWindow" class anymore), the *find_base_move ()* function and the animation functions, as well as some access functions (to get the sizes needed outside of this class), or the initialisation function.

```
class CBodySize : public CAnimAngles
{
private: //31 values
float FOOT_JOINT_SIZE,
       FOOT_HEIGHT,
      FOOT_WIDTH,
      FOOT,
      UP_ARM_HEIGHT,
      UP_ARM_WIDTH,
```

```
    UP_ARM_JOINT_SIZE,
    LO_ARM_HEIGHT,
    LO_ARM_WIDTH ,
    LO_ARM_JOINT_SIZE ,
    HAND_HEIGHT,
    HAND_WIDTH,
    HAND,
    FINGER_SIZE,
    TORSO_WIDTH,
    TORSO_HEIGHT,
    TORSO,
    HEAD_WIDTH,
    HEAD_HEIGHT,
    HEAD_JOINT_SIZE,
    BASE_WIDTH,
    BASE_HEIGHT,
    UP_LEG_HEIGHT,
    UP_LEG_JOINT_SIZE,
    UP_LEG_WIDTH,
    LO_LEG_HEIGHT,
    LO_LEG_WIDTH,
    LO_LEG_JOINT_SIZE,
    LEG_HEIGHT,
    TORSO_JOINT_SIZE,
    HAND_JOINT_SIZE;

public:
    float walking_angles[2][6] ;
    int FRAMES;
    double base_move;
    float langle_count,
    langle_count2,
    rangle_count,
    rangle_count2;

    CBodySize();
    virtual ~CBodySize(){}; // the destructor does nothing
```

```
    void InitialSize(void);
    void SaisieSize(void);

//to access private data from outside=more secure like that than to have public data.
    float AccessLoArm(void){return LO_ARM_HEIGHT;}
    float AccessUpArm(void){return UP_ARM_HEIGHT;}
    float AccessTorso(void){return TORSO_HEIGHT;}
    float AccessLegH(void){return LEG_HEIGHT;} //used for the zoom
    float AccessULJS(void){return UP_LEG_JOINT_SIZE;} //used for the zoom

//drawing functions
    void DrawCube (float, float, float);
    void Draw_Model(void);
    void Draw_Head(void);
    void Draw_Torso(void);
    void Draw_Upper_Arm(void);
    void Draw_Lower_Arm(void);
    void Draw_Hand(void);
    void Draw_Arm(int);
    void Draw_Torso_Model(void);
    void Draw_Base(void);
    void Draw_Upper_Leg(void);
    void Draw_Lower_Leg(void);
    void Draw_Foot(void);
    void Draw_Leg(int);
    void Draw_Base_Legs(void);

    void animate_body(CAnimAngles angles[4]);
    double find_base_move(double langle_up, double langle_lo, double rangle_up, double rangle_lo);
};
```

5.1.2 Hierarchy and drawing of the model

Once again, it is necessary to build the model in a hierarchical way (Figure 20).

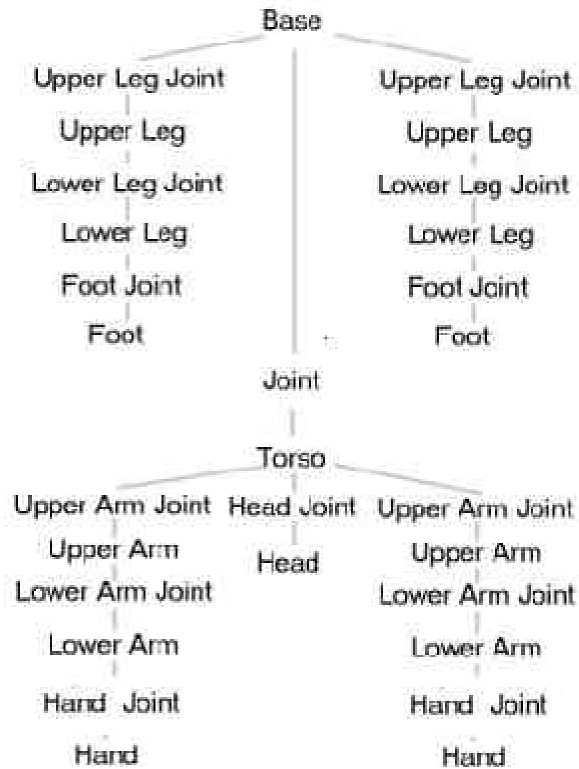


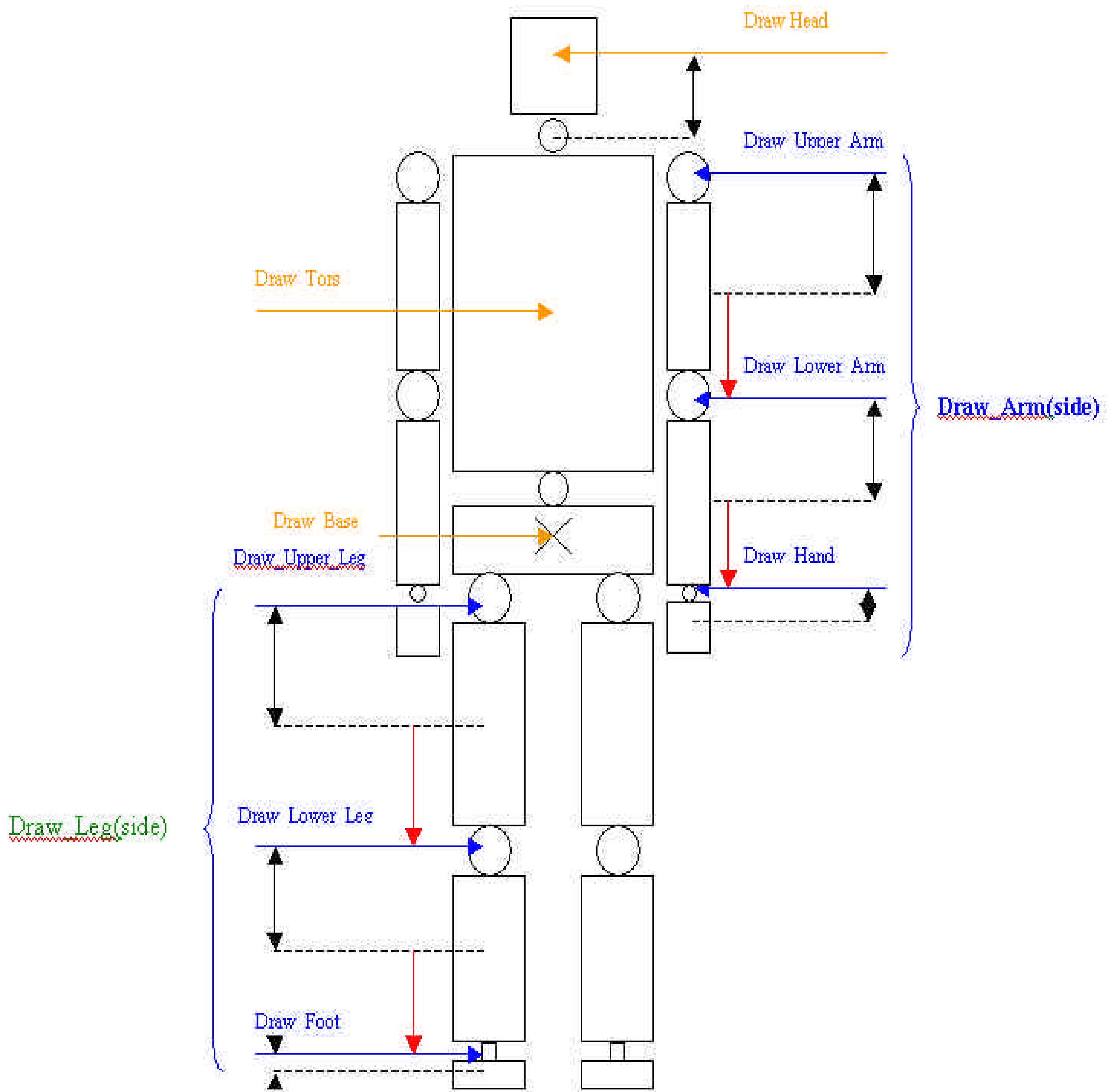
Figure 20. Model hierarchy

The code of the *Draw_Model ()* function which draws the complete model of the man is explained in detail below. A call to this function is sufficient to draw the complete model.

```
void CBodySize::Draw_Model(void)
{
    glPushMatrix(); // The matrix prior to this function is saved
    glTranslatef(0.0,base_move,0.0) ;

    // The base is created
    Draw_Base() ;
    // The matrix is saved again to place the second in the hierarchy, the torso joint
    glPushMatrix();
```

```
// The centre of the coordinates is moved to the correct place and the joint is created
glTranslatef(0.0,TORSO_JOINT_SIZE+BASE_HEIGHT/2,0.0) ;
// Draw the torso joint
glPushMatrix() ; // Save the matrix before changing the axes scale
glScalef(TORSO_JOINT_SIZE,TORSO_JOINT_SIZE,TORSO_JOINT_SIZE) ;
glColor3f(0.8,0.8,0.8) ;
glutSolidSphere(1.0,8,8);
glPopMatrix() ; // Restore the original axes scale

// The center of the coordinates is moved to the correct place and the torso is created
glTranslatef(0.0,TORSO_JOINT_SIZE,0.0) ;
// Equivalent to Draw_Torso( third in the hierarchy), Draw_Head( joint + head, fourth and fifth in the
hierarchy), Draw_Arm(LEFT), Draw_Arm(RIGHT)
// Appropriated save of matrix are done into these functions body.
Draw_Torso_Model();

glPopMatrix();

// Equivalent to Draw_Led(LEFT), Draw_Leg(RIGHT), same kind of  level of hierarchy as the arms.
Draw_Base_Legs();

glPopMatrix();
}
```

The relations existing between the drawing functions are shown on the Figure 21 below.



Translation

Duration of the function

Figure 21. Relations in the model

5.2 Animating the model

5.2.1 Walking Animation implementation

Using the revised approach, the animation angles are read from a file instead of being hard coded in the walking procedure. A class called "CAnimAngles" is built in order to store the animation angles read from the file into its 15 float members. There is therefore a new function allowing the program to read into a file the angles and to store them into a tab of CAnimAngles objects:

*Read_Data_From_File (CAnimAngles *init, CAnimAngles array[])*).

This function accepts two arguments of type CAnimAngles. The first one is a pointer and is used to store the initial (prior to animation) angles. The second argument is an array of four arguments (=half the number of key frames used in this model) where the angles of the first four key frames are stored. The file in which the animation angles are read is called 'data.txt', and contains the following values (these values may be changed to get another walk or to get the body sitting or jumping etc…):

5 lines

```
0.0 0.0 0.0  40.0   -8.0  50.0 -30.0   0.0 0.0 -40.0  -8.0    0.0   30.0  0.0 0.0
0.0 0.0 0.0 -25.0    0.0  10.0  15.0  15.0 0.0  20.0 -17.0  -30.0    5.0  5.0 0.0
0.0 0.0 0.0 -15.0  -32.0 -10.0  20.0 -15.0 0.0  20.0 -15.0   80.0  -65.0 85.0 0.0
0.0 0.0 0.0 -20.0   15.0 -80.0   5.0  10.0 0.0  15.0  32.0  -10.0  -25.0 -40.0 0.0
0.0 0.0 0.0 -20.0   17.0  30.0  20.0 -10.0 0.0  25.0   0.0   10.0   25.0 -50.0 0.0
```

15 columns

This file is composed of 5 lines of 15 numbers each (related to the 15 private members of the CAnimAngles class) and it is related to the walking cycle. The first line contains the initial angles, and the next four the angles of the first four key frames. These values are given to the program for the *animate_function()* via the *Read_Data_From_File()* function.

The class CAnimAngles contains 15 private floats (corresponding to the animation angles) to store the data read from this file.

```
class CAnimAngles
{
public:
        // Each of these elements is aimed to store the animation angle for the particular body part.
                        // Values between two key frames(or the initial position and the first one)
        float head;             //1  ✍   0,    0,    0 ,   0,    0
        float upbody  ;         //2  ✍   0,    0,    0,    0,    0
        float lobody  ;         //3  ✍   0,    0,    0,    0,    0
        float l_uparm ;         //4  ✍  40, -25, -15, -20, -20
        float l_loarm ;         //5  ✍  -8,    0, -32,  15,  17
        float l_hand  ;         //6  ✍  50,  10, -10, -80,  30
        float l_upleg ;         //7  ✍ -30,  15,   20,    5,  20
        float l_loleg ;         //8  ✍   0,  15, -15,  10, -10
        float l_foot  ;         //9  ✍   0,    0,    0,    0,    0
        float r_uparm ;         //10 ✍  -40,  20,  20,  15,  25
        float r_loarm ;         //11 ✍  -8, -17, -15,  32,   0
        float r_hand  ;         //12 ✍  0, -30,  80, -10,  10
        float r_upleg ;         //13 ✍ 30,   5, -65, -25,  25
        float r_loleg ;         //14 ✍   0,   5,  85, -40, -50
        float r_foot  ;         //15 ✍   0,   0,    0,    0,   0

        CAnimAngles();                  // the constructor does nothing.
        virtual ~CAnimAngles(){};               // the destructor does nothing
};
```

Therefore, in each line of the file, the first value corresponds to the head angle, the second one to the up_body angle, and so on until the last one (15th) corresponding to the right foot angle. To illustrate this, if the init angle entered is 30 degrees for the left upper leg, and –30 for the right one. According that a negative value is in the positive direction and that a positive one corresponds to a back position of the corresponding limb, the right leg is behind from 30 degrees while the right one is in front from 30 degrees (Figure 22).

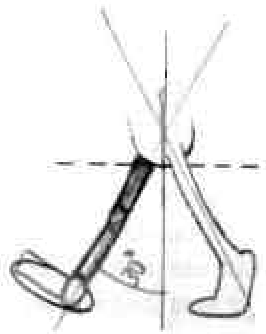
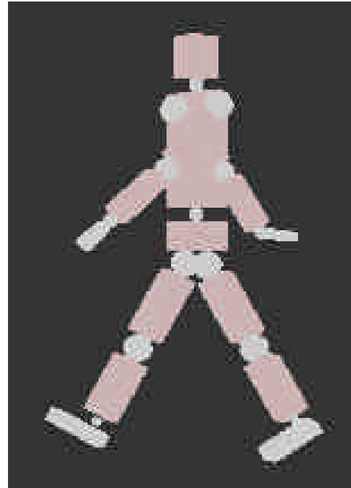



Figure 22. Initial upper leg angles

The structure of this function uses *fscanf* calls to read the angles from the file and places them in one of two previously mentioned variables. An integer variable 'counter' is also used to count how many values are actually read from the file.

```
for (counter = 0 ; counter < 4 ; counter++)
{
        scan_counter += fscanf(working_file,"%f",&array[counter].head) ;
        scan_counter += fscanf(working_file,"%f",&array[counter].upbody) ;
        scan_counter += fscanf(working_file,"%f",&array[counter].lobody) ;
        --------------- And so on ----------------------------------------
}
```

The angles are read into the CMainWindow constructor, and then used via the array by the function *animate_body ()* or used to initialise the body position. The function Read_data_From_File () uses the first line of 'data.txt' to fill in its first argument '&init_angle': the values read in this first line are used to initialise the human body position in the constructor of CMainWindow.

```
// Initialise the body position
Human.walking_angles[LEFT][0]  = init_angles.l_uparm ;
Human.walking_angles[RIGHT][0] = init_angles.r_uparm ;
Human.walking_angles[LEFT][1]  = init_angles.l_loarm ;
Human.walking_angles[RIGHT][1] = init_angles.r_loarm ;
Human.walking_angles[LEFT][2]  = init_angles.l_hand ;
Human.walking_angles[RIGHT][2] = init_angles.r_hand ;
Human.walking_angles[LEFT][3]  = init_angles.l_upleg ;
Human.walking_angles[RIGHT][3] = init_angles.r_upleg ;

Human.walking_angles[LEFT][4]  = init_angles.l_loleg ;
Human.walking_angles[RIGHT][4] = init_angles.r_loleg ;
Human.walking_angles[LEFT][5]  = init_angles.l_foot ;
Human.walking_angles[RIGHT][5]  = init_angles.r_foot ;
```

The 4 next lines of 'data.txt' are used to store the animation angles themselves, in the CAnimObject 'angles'. They only represent the first four key frames (corresponding to half the total number of key frames used in here). As the key frames are symmetrical (the last four to the first four) the values of this array are also used to calculate the angles of the last four key frames. The last four key frames make use of the angles values of the first four because the assumed walk is symmetrical. Therefore, the case (5) in the switch loop of the animate_body () function corresponds to the symmetrical of the case (1), as it is illustrated by the code bellow.

```
// The key frame side for the angle is the same as the side used for the animation between the init angles
and the ones of the first key frame.
case 1 :
        l_upleg_add = angles[0].l_upleg / FRAMES ;
        r_upleg_add = angles[0].r_upleg / FRAMES ;
        --- and so on for the other angles ---------------
```

is identical to the case 5, with the only distinction being that the left side angles of the first key frame are used for the right side of the fifth key frame (and identically the right side angles are used for the left side of the animation)

```
// The first key frame side for the angles are the opposite of  the ones used for the
```

```
// animation between key frame 4 and key frame 5.
case 5 :
        l_upleg_add = angles[0].r_upleg / FRAMES ;
        r_upleg_add = angles[0].l_upleg / FRAMES ;
      --- and so on for the other angles --------------
```

The animation function *animate_body ()* is similar to the one used with the bottom model. The main difference being that now, using the same technique as the one previously explained for the legs, it also calculates the walking angles of the arms. It makes use of array *angles[4]* of four CAnimAngles objects, filled in by the *Read_data_From_File ()* function, and containing the angles for the first four key frames; *angles[0]* contains the angle between the initial position and the first key frame, *angles[1]* the ones between the first and the second key frames, and so on until *angles[3]* that contains the angles of the body parts between the third and fourth key frames. *l_upleg_add, l_loleg_add, r_upleg_add, r_loleg_add* are calculated by dividing the initial angles difference (between the two key frames) by the number of frames that are needed to make the animation (number of frames between two key frames). These are the rotation values for a single frame animation. The number of needed frames between two key frames is initially set to 20, but this value can dynamically be changed using the menu. The values of the previously calculated variables are stored in the *walking_angles[2][6]* array. They are used while drawing the model to rotate the different parts of the body and to animate it.

Walking_angles [side][0] contain the angle for the upper arms,

Walking_angles [side][1] the ones for the lower arms,

Walking_angles [side][2] the ones for the hands,

Walking_angles [side][3] the ones for the upper legs,

Walking_angles [side][4] the ones for the lower legs,

Walking_angles [side][5] the ones for the feet,

with *side* being either RIGHT or LEFT to specify which side is being rotated.

The *l_upleg_add*, *l_loleg_add*, *r_upleg_add*, *r_loleg_add* values are then added to the variables *langle_count* (<-> *langle_up* in the *base_move* function), *langle_count2* (<-

>*langle_lo*), *rangle_count* (<-> *rangle_up*) and *rangle_count2* (<-> *rangle_lo*). That's illustrated on Figure 23.

If ? is the upper leg angle at the key frame z, and r the displacement assumed between frame z and frame (z+1), then the upper leg angle at frame (z+1) is ?'=? + r.

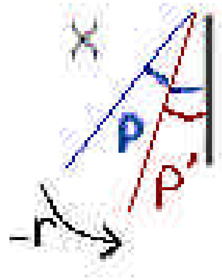


Figure 23. Angles for the body's vertical displacement

Indeed a positive upper leg angle corresponds to a leg at the back of the body. To bring the leg back to the front, the angle entered into 'data.txt' is positive, so if the leg is at –30 degrees, and we bring it back to –25 degrees, the value read in *angle[x]* is positive and equal to 5, giving ?'=-30+5=-25?, which is the expected result. Identically, if the leg is in front of the body, with an angle of 30 degrees (as an example), and if the angle read in 'data.txt' is –5 degrees, that means that at the next key frame is situated at ?' = 30 + (-5) = 25?. The simple case of the left upper arm angles also provides a good illustration of this. Its initial angle is negative and equal to -40?, so the arm is in front of the body. During the next four key frames, it is moved to the back of the body, therefore the animation angles read into 'data.txt' are all positive for the next four key frames: 20, 20, 15, 25.

These angles are only used within the *animate_body ()* function to find the movement of the base calling *find_base_move ()*. A part of the code is given below in order to show the way all this is implemented.

```
void CBodySize::animate_body(CAnimAngles angles[4])
{
static frames = FRAMES,  // Static since they only need to be initialised once.
        flag = 1 ;                  // Initialised to one so that the switch starts with the case 1.

  float  l_upleg_dif ,
           r_upleg_dif ,
           l_upleg_add ,
           r_upleg_add ,
           l_loleg_dif ,
           r_loleg_dif ,
           l_loleg_add ,
           r_loleg_add ;

// Skeleton of the function. All the operations needed to be done for the walking animation  are done in this
// statement. It depends on the variable flag, initially set to 1.

Switch(case)
{
 case 1 :
// These are the rotation values for a single frame animation
          l_upleg_add = angles[0].l_upleg / FRAMES ;
          r_upleg_add = angles[0].r_upleg / FRAMES ;
          l_loleg_add = angles[0].l_loleg / FRAMES ;
          r_loleg_add = angles[0].r_loleg / FRAMES ;

          l_uparm_add = angles[0].l_uparm / FRAMES ;
          l_loarm_add = angles[0].l_loarm / FRAMES ;
          l_hand_add  = angles[0].l_hand / FRAMES ;
          r_uparm_add = angles[0].r_uparm / FRAMES ;
          r_hand_add  = angles[0].r_hand / FRAMES ;
          r_loarm_add = angles[0].r_loarm / FRAMES ;

// The walking_angles array is used in the draw_base function in order to animate the model.
// The values of the variable are note just copied: they are added to the previous value of the array in order
// that the array contains the value for the next frame. This is done because the rotation is not
// incremental.
```

```
        walking_angles[LEFT][3] += l_upleg_add ;
        walking_angles[RIGHT][3] += r_upleg_add ;
        walking_angles[LEFT][4] += l_loleg_add ;
        walking_angles[RIGHT][4] += r_loleg_add ;

        walking_angles[LEFT][0] += l_uparm_add ;
        walking_angles[LEFT][1] += l_loarm_add ;
        walking_angles[LEFT][2] += l_hand_add ;
        walking_angles[RIGHT][0] += r_uparm_add ;
        walking_angles[RIGHT][1] += r_loarm_add ;
        walking_angles[RIGHT][2] += r_hand_add ;

// These values are then added to  the variable langle_count ... which contain the initial value
// of the angles of the bottom parts. They is going to be used with the function base_move in
// order to calculate the body's vertical displacement.
        langle_count  += l_upleg_add ;
        langle_count2 += l_loleg_add ;
        rangle_count  += r_upleg_add ;
        rangle_count2 += r_loleg_add ;

        base_move = find_base_move(langle_count,langle_count2,rangle_count,rangle_count2) ;
        frames-- ;
        if (frames == 0) // The second key frame is reached, this is the end of the first cycle
        {
                flag = 2 ;                    // The variable flag is incremented
                frames = FRAMES ;             // Frames is reinitialised to FRAMES
        }

------------------------------and so on until case 8 ----------------------------------
}
```

The values of the variables are not just copied but they are added to the previous values of the array so that the array then contains the angles for the next frame. This is done because the rotation is not incremental. OpenGL follows the non-incremental technique in order to diminish cumulative errors that may appear if this particular modelling transformation would be based on an incremental technique. For example if the function

glRotatef(-30, 1.0, 0.0, 0.0) is used to rotate the upper leg by twenty degrees and at the next step the upper leg is needed to be rotated by five degrees more, the correct call is *glRotatef(-25, 1.0, 0.0, 0.0)* and not *glRotatef(5, 1.0, 0.0, 0.0)*. According that the initial angle of the right upper leg, as can be read in 'data.txt', is –30 degrees, and that the angle between this angle and the first key frame is 5, it means that the angle of the leg at the first key frame is then –30+5=-25 degrees, as illustrated on the Figure 24.

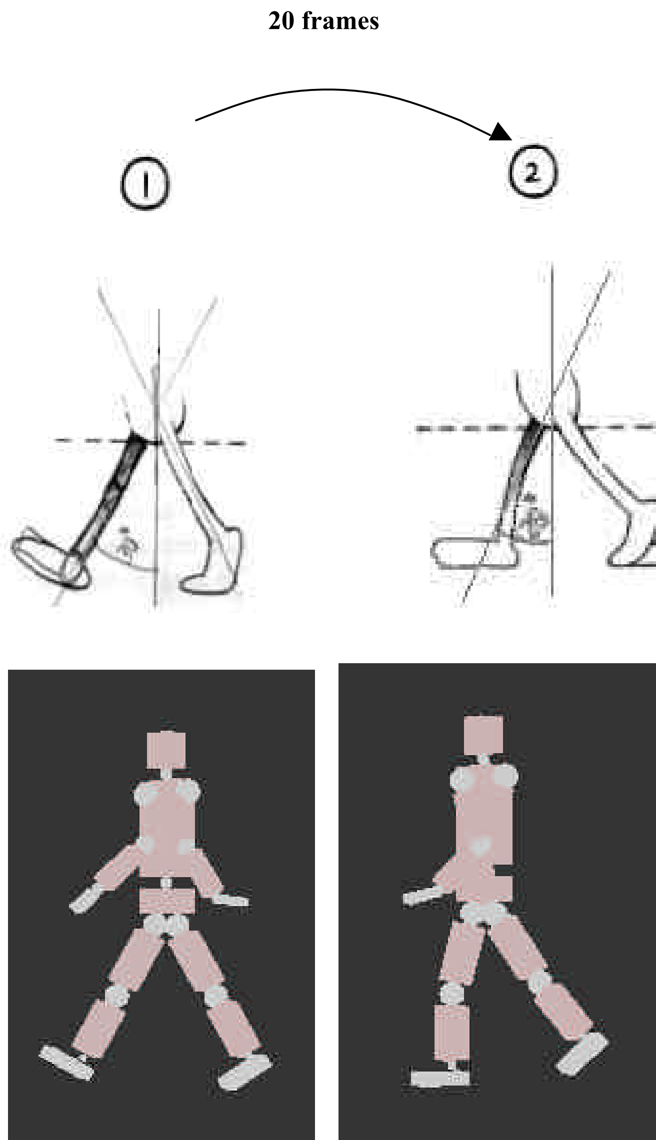


Figure 24. Rotation of the leg

An example of the way the angles stored in the walking_angles are used is illustrated below by making use of the *Draw_Arm ()* and *Draw_Leg ()* functions.

```
void CBodySize::Draw_Arm(int side)
{
        glPushMatrix() ;

        glRotatef(walking_angles[side][0],1.0,0.0,0.0) ;
        Draw_Upper_Arm() ;

        glTranslatef(0.0,- (UP_ARM_HEIGHT/2 + LO_ARM_JOINT_SIZE),0.0) ;

        glRotatef(walking_angles[side][1],1.0,0.0,0.0) ;
        Draw_Lower_Arm() ;

        glTranslatef(0.0,- (LO_ARM_HEIGHT/2+HAND_JOINT_SIZE ), 0.0) ;

        glRotatef(walking_angles[side][2],1.0,0.0,0.0) ;
        Draw_Hand() ;

        glPopMatrix() ;
}

void CBodySize::Draw_Leg(int side)
{
  glPushMatrix() ;

  glRotatef(walking_angles[side][3],1.0,0.0,0.0) ;
  Draw_Upper_Leg() ;

  glTranslatef(0.0,- (UP_LEG_HEIGHT/2 + LO_LEG_JOINT_SIZE),0.0) ;
  glRotatef(walking_angles[side][4],1.0,0.0,0.0) ;
  Draw_Lower_Leg() ;

  glTranslatef(0.0,- (LO_LEG_HEIGHT/2 + FOOT_JOINT_SIZE), 0.0) ;
  glRotatef(walking_angles[side][5],1.0,0.0,0.0) ;
  Draw_Foot() ;
```

glPopMatrix()

}

5.2.2 Menu to act on the model

The user has the ability to act on the human model via the menu[13][14].

The paragraph 5.2.4 below explains in more detail how to add an item to the menu using the particular case of the 'Human Size…' item, knowing that the same method can be used to create every item of the menu. The application menu is mainly composed of three main items, each of them providing a set of functionality. The way it works is described below and is illustrated in Figure 25, Figure 26 and Figure 27.

In the File menu, if the item 'Rotation On' is selected, the user can then ever choose to rotate its model Faster or Slower. In the case the model is animated ('Animation' checked), the number of frames can then be increased or decreased, giving more or less precision to the movement (as it is explained earlier). It's only in the case where there is no stereo that it is possible to 'Zoom in' or 'Zoom out'. It would be feasible to add this functionality with the stereo on, but then the user would need to adapt the magnitude and parallax parameters to its sight.

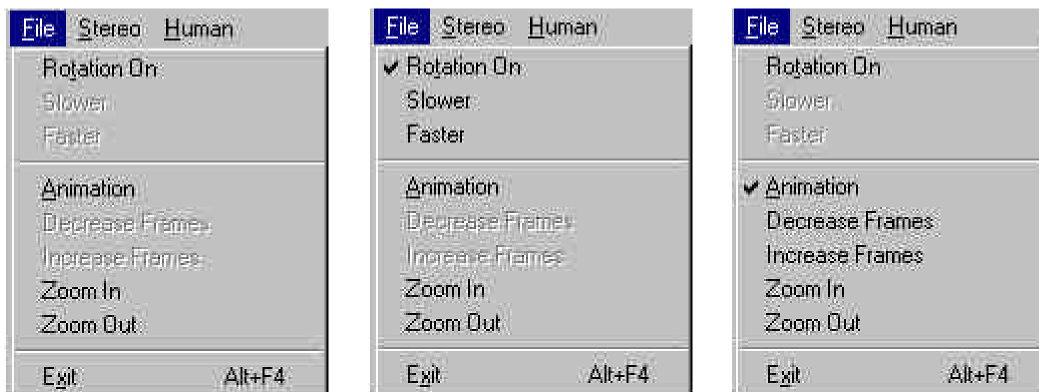


Figure 25. 'File' menu

In the Stereo menu, if the user checks 'Stereo On' (which adds the stereo effects to the human model), then he can adapt the magnitude and parallax parameters according to its

own view. Indeed, the distance between human eyes can vary from a person to another, as it is explained in chapter 2.

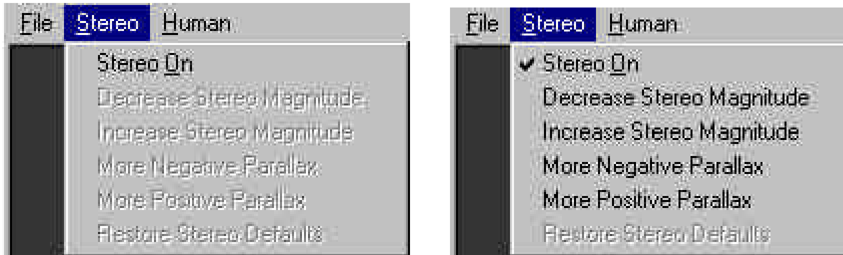
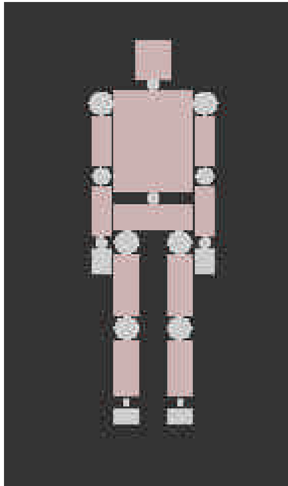


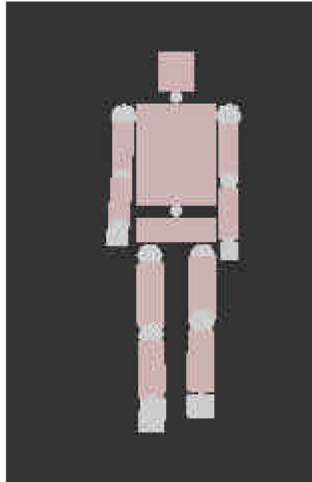
Figure 26. 'Stereo' menu

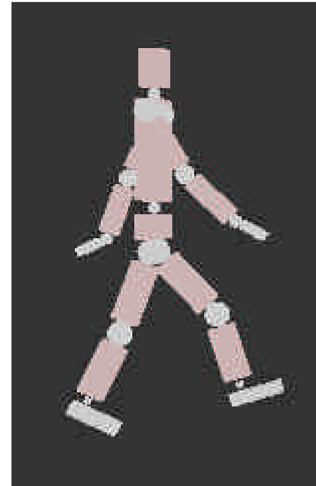
In the Human menu, selecting ‘Change Size…’ opens a Dialog Box into which the user can enter new parameters for the Human body, as well as a name for its model. The way the dialog box is built is explained in more detail in the paragraph 5.3 of this Chapter. Selecting ‘Restore Default Size’ redraw the human model with the default size.

Figure 27. 'Human' menu

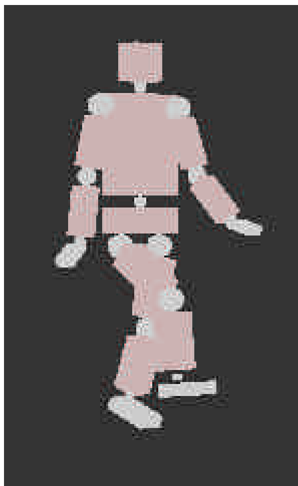
After compiling and running the program, the user can see the entire model walk, as shown on the screen shots taken from this program and shown on Figure 28. It is possible to leave the human model static and/or rotate it and/or animate it etc… using the menu.

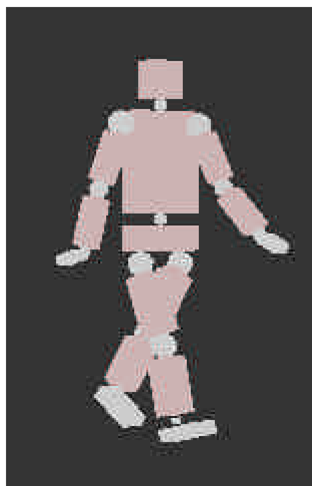


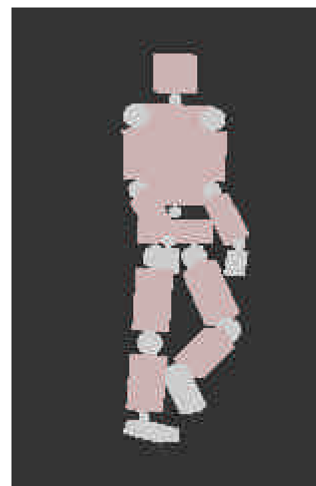




Front Static model **Front Walking model** **Profile Walking model**







Semi-Profile Walking model

Figure 28. The "Animated" Human Model

5.3 Testing the walking model

5.3.1 Coding tests

This software has been written in order to check the main possible errors. First of all, the way stereo is tested is already explained in the section 2.2.1 of Chapter 2. Mainly, a

message box is displayed informing the user that the current display configuration does not support OpenGL stereo, and the program quits after having killed the application. Other tests are necessary, and they are all done using the *EnableStereo ()* function (which is a member function of the CMainWindow class): tests are done in this function and the return value FALSE in the case there is a problem and display message box in the main time to prevent the user that there is a problem. Then when calling the *EnableStereo ()* function during the initialisation (in the *InitInstance ()* function of the CMyApp Class), if the return value is effectively FALSE, then the program quits. The different tests done are described below using pieces of code of the human model application.

First of all the program checks if it managed to get a Device Context

```
if (hdc==NULL)
{
        AfxMessageBox (_T("Can't Create A GL Device Context."));
        return FALSE;       // The window is then killed in InitInstance, since the return value is FALSE
}
```

It then checks if Windows managed to find a matching pixel format

```
if (iPixelFormat==0)
{
        AfxMessageBox (_T("Can't Find a suitable Pixel Format."));
        return FALSE;
}
```

Before it checks if the pixel format is settable.

```
BOOL bSuccess = SetPixelFormat(hdc, iPixelFormat, &pfd);
if( !bSuccess
{
        AfxMessageBox (_T("Can't Find a suitable Pixel Format."));
        return FALSE;
}
```

While getting a rendering context, it checks if the rendering context can be got.

```
hGLRC = wglCreateContext (hdc);
if(hGLRC==0)
```

```
{
	AfxMessageBox (_T("Can't Create A GL Rendering Context."));
	return FALSE;
}
```

The program then verifies if it's possible to activate the rendering context.

```
bSuccess = wglMakeCurrent (hdc, hGLRC); // Try To Activate The Rendering Context if (bSuccess==0)
if(hGLRC==0)
{
	AfxMessageBox (_T("Can't Activate The GL Rendering Context."));
	return FALSE;
}
```

All these error checking do return FALSE in the case a problem is found. It is then useful while initialising the application. Indeed, the *InitInstance ()* function contains the following lines, allowing quitting the application.

```
//error checking
if (!((CMainWindow *)m_pMainWnd)->EnableStereoWindow())
{
	m_pMainWnd->ShowWindow (SW_HIDE);
	m_pMainWnd->DestroyWindow();
	exit(0);
}
```

There is also a test provided while trying to read the angles, in order to be sure that the data file is available.

```
if (working_file == NULL)
{
	AfxMessageBox (_T("Searched the following directories and could not find the data file."));
	exit(0) ;
}
```

The function *Write_Test_Data ()* is implemented for testing reasons. The purpose of this function is to be sure that the animation angles in the file 'data.txt' are correctly read. It therefore prints these angles into the 'test.txt' file. This function accepts the same two

arguments as the function described above, but instead of initialising them it uses them to output their values into a file, for testing reasons.

Write_Test_Data (CAnimAngles init_data, CAnimAngles data[]).

Then by comparing the two files 'data.txt' and 'test.txt' a user can find out if the program reads in the correct animation angles. Below are both of these files. Their similarities provides the security that these data are correctly read.

Data.txt:

0.0 0.0 0.0 40.0 -8.0 50.0 -30.0 0.0 0.0 -40.0 -8.0 0.0 30.0 0.0 0.0

0.0 0.0 0.0 -25.0 0.0 10.0 15.0 15.0 0.0 20.0 -17.0 -30.0 5.0 5.0 0.0

0.0 0.0 0.0 -15.0 -32.0 -10.0 20.0 -15.0 0.0 20.0 -15.0 80.0 -65.0 85.0 0.0

0.0 0.0 0.0 -20.0 15.0 -80.0 5.0 10.0 0.0 15.0 32.0 -10.0 -25.0 -40.0 0.0

0.0 0.0 0.0 -20.0 17.0 30.0 20.0 -10.0 0.0 25.0 0.0 10.0 25.0 -50.0 0.0

Test.txt:

Test File for checking if input data are correct

0.0 0.0 0.0 40.0 -8.0 50.0 -30.0 0.0 0.0 -40.0 -8.0 0.0 30.0 0.0 0.0

0.0 0.0 0.0 -25.0 0.0 10.0 15.0 15.0 0.0 20.0 -17.0 -30.0 5.0 5.0 0.0

0.0 0.0 0.0 -15.0 -32.0 -10.0 20.0 -15.0 0.0 20.0 -15.0 80.0 -65.0 85.0 0.0

0.0 0.0 0.0 -20.0 15.0 -80.0 5.0 10.0 0.0 15.0 32.0 -10.0 -25.0 -40.0 0.0

0.0 0.0 0.0 -20.0 17.0 30.0 20.0 -10.0 0.0 25.0 0.0 10.0 25.0 -50.0 0.0

Apart from the tests made all along the code, visual tests do also present an interest.

5.3.2 Visual tests

It's interesting to visualize if a change in the 'data.txt' field corresponds to the correct change into the model motion, and it's also useful to compare the human model walking with a real human walking, in order to see if the motion chosen is realistic. So that's what is done in this section.

Changes are made among the angles of the 'data.txt' file in order to animate the human in a different way. As an example changing the angles from different parts of the body at different times allowed to check if the correct parts were animated as expected. Another

example is that the upper legs amplitude and the upper arms angles are divided by two , this results in a smaller amplitude of movement, which is the expected result 5 (Figure 29, Figure 30).

New file entered:

0.0 0.0 0.0 ***20.0*** *-8.0 50.0* ***-15.0*** *0.0 0.0* ***-20.0*** *-8.0 0.0* ***15.0*** *0.0 0.0*

0.0 0.0 0.0 ***-12.0*** *0.0 10.0* ***7.0*** *15.0 0.0* ***10.0*** *-17.0 -30.0* ***2.5*** *5.0 0.0*

0.0 0.0 0.0 ***-7.0*** *-32.0 -10.0* ***10.0*** *-15.0 0.0* ***10.0*** *-15.0 80.0* ***-32.0*** *85.0 0.0*

0.0 0.0 0.0 ***-10.0*** *15.0 -80.0* ***2.5*** *10.0 0.0* ***7.0*** *32.0 -10.0* ***-12.0*** *-40.0 0.0*

0.0 0.0 0.0 ***-10.0*** *17.0 30.0* ***10.0*** *-10.0 0.0* ***12.0*** *0.0 10.0* ***12.0*** *-50.0 0.0*

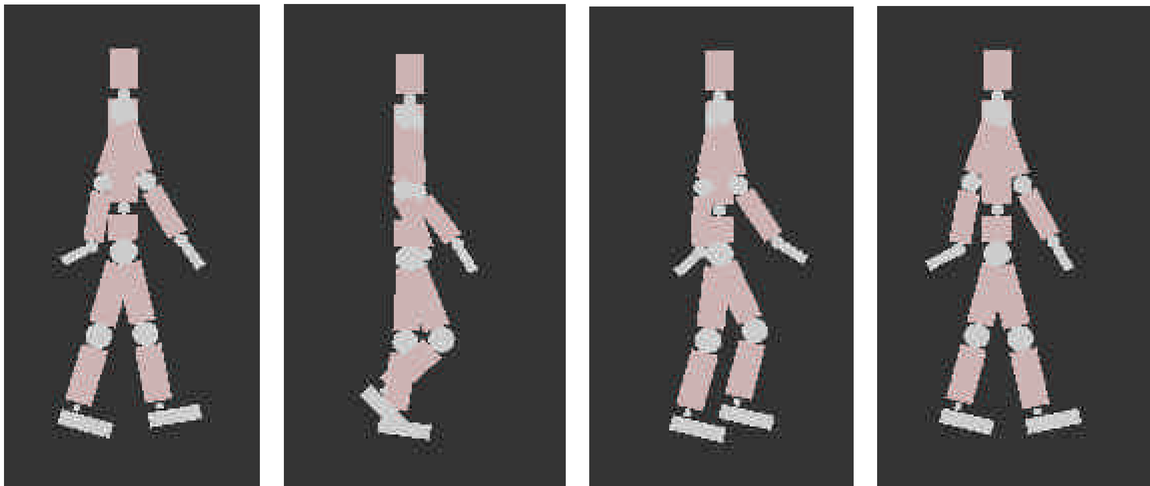


Figure 29. Smaller amplitudes

Instead of the initial angles entered:

0.0 0.0 0.0 ***40.0*** *-8.0 50.0* ***-30.0*** *0.0 0.0* ***-40.0*** *-8.0 0.0* ***30.0*** *0.0 0.0*

0.0 0.0 0.0 ***-25.0*** *0.0 10.0* ***15.0*** *15.0 0.0* ***20.0*** *-17.0 -30.0* ***5.0*** *5.0 0.0*

0.0 0.0 0.0 ***-15.0*** *-32.0 -10.0* ***20.0*** *-15.0 0.0* ***20.0*** *-15.0 80.0* ***-65.0*** *85.0 0.0*

0.0 0.0 0.0 ***-20.0*** *15.0 -80.0* ***5.0*** *10.0 0.0* ***15.0*** *32.0 -10.0* ***-25.0*** *-40.0 0.0*

0.0 0.0 0.0 ***-20.0*** *17.0 30.0* ***20.0*** *-10.0 0.0* ***25.0*** *0.0 10.0* ***25.0*** *-50.0 0.0*

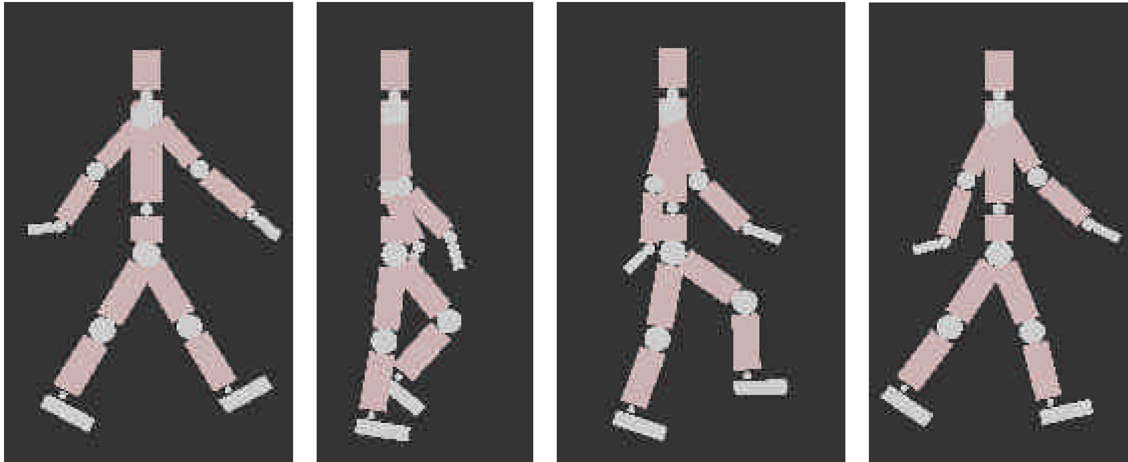
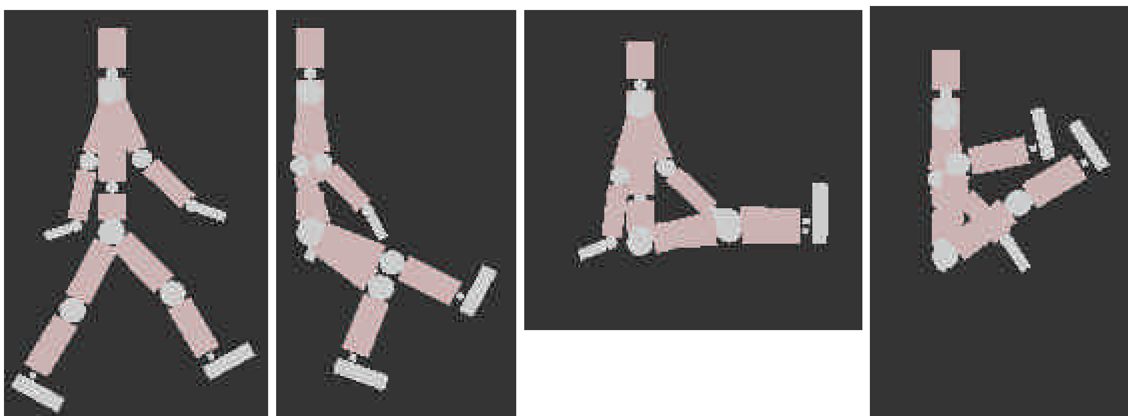


Figure 30. Initial Amplitudes

Since no constraints are added to the human model angles at the moment, it is possible to enter any values for the angles and therefore to animate the model in an abnormal manner, without any complaint of the program. A good way to have correct angles is to use a databank of angles taken from a real human, using motion capture, as it is explained in Chapter 6. Below (Figure 31) is an example of the abnormal manner in which the human model can be animated at the moment.



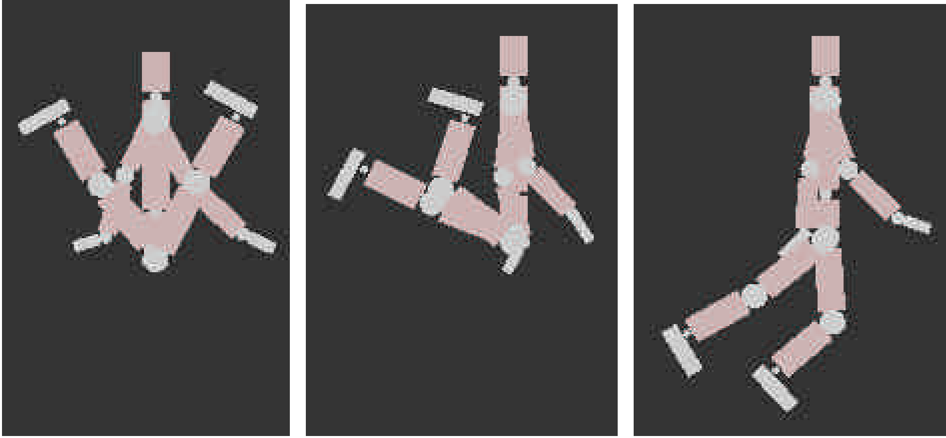


Figure 31. Abnormal animation of the model

By using this 'data.txt' file:

0.0 0.0 0.0 40.0 -8.0 50.0 -30.0 0.0 0.0 -40.0 -8.0 0.0 30.0 0.0 0.0
0.0 0.0 0.0 -25.0 0.0 10.0 -15.0 15.0 0.0 20.0 -17.0 -30.0 -5.0 5.0 0.0
0.0 0.0 0.0 -15.0 -32.0 -10.0 -20.0 -15.0 0.0 20.0 -15.0 80.0 -65.0 85.0 0.0
0.0 0.0 0.0 -20.0 15.0 -80.0 -5.0 10.0 0.0 15.0 32.0 -10.0 -25.0 -40.0 0.0
0.0 0.0 0.0 -20.0 17.0 30.0 -20.0 -10.0 0.0 25.0 0.0 10.0 -25.0 -50.0 0.0

And this example is an example among many others.

By comparing the way the motion happens in the case of a real human (Figure 32) and in the case of the human model application (Figure 33), it is also possible to test and validate the human motion actually chosen (or the angles chosen for the walking animation).

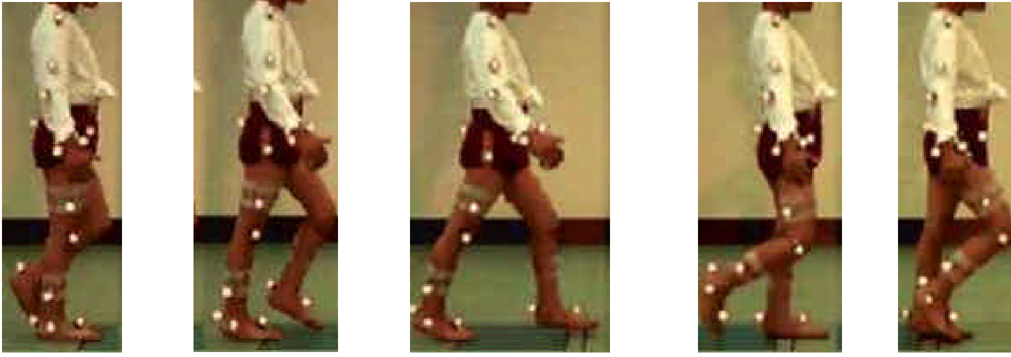


Figure 32. Real human walking

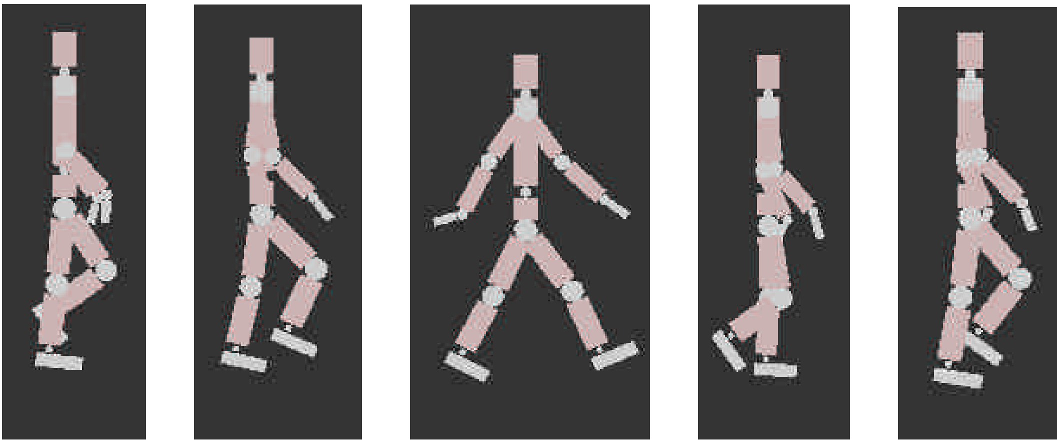


Figure 33. Human model application walking

5.4 Sizing the model

As the point of interest in this application is more the hierarchical model than the accuracy of the size, the heights and widths of the body are based on a not so accurate hand-drawn sketch of a man. Indeed, the user can enter each of these values dynamically via a dialog box called from the menu, allowing the sizes corresponding exactly to the person he wants to train to be entered, without worrying about the default values. This is possible because there is a class for the model itself instead of define values.

In this section the main steps to follow to call the ‘Human Size…’ dialog box from the menu are explained. The procedure is exactly the same to create a different dialog box: it’s then just necessary to choose different names and to define the wanted functionalities.

5.4.1 Adding an item to the menu

✍ Using the Interface of Visual C++ (resource), first add a new item into the menu, named ‘Human Size…’. The “…” corresponds to the usual convention to let the user know that a dialog box can be displayed from here.

✍ When saving, it automatically declares an ID value for this new item into the header resource file. In the human model application case, this name is redefined (even if it’s not necessary), giving:

```
#define IDM_FILE_RESTORESIZE          40001
```

✍ In the message map, two functions necessary for the menu are then added, correlated to this new item ID:

```
ON_COMMAND (IDM_FILE_RESTORESIZE, OnRestoreSize)
ON_UPDATE_COMMAND_UI (IDM_FILE_RESTORESIZE, OnRestoreSizeUpdate)
```

✍ It is then necessary to define what these functions do, since they are called when the user selects the “Human Size…” item from the menu. Basically a call to the *SaisieSize ()* function is made. This function defines in its body what is done when the user selects the ‘Human Size…’ item. What the *SaisieSize ()* function does is defined separately (in this application case, it calls the dialog box, since that’s what is supposed to be done when the user select the “Human Size …” item. That is however seen in more detail later on). This function is a member function of the “CbodySize” class, that’s why it is called by doing *Human.SaisieSize()*, since it refers to the Human object of this class.

```
void CMainWindow::OnHumanNewSize ()
{
        if(!bHumanOn)
```

```
    {
            Human.SaisieSize();  // Function called when the user click on the 'Human Size…' item
            bHumanOn=!bHumanOn;
            Invalidate();
    }
    else
            bHumanOn=!bHumanOn; // Better: we go back to the old configuration
}
void CMainWindow::OnHumanNewSizeUpdate (CCmdUI* pCmdUI)
{
        pCmdUI->SetCheck (bHumanOn); // Menu item active
}
```

5.4.2 The Dialog Box

If the aim is to display a dialog box from an MFC application, it is easy to find lots of information on the Internet on this subject. Anyway, it's much harder to find documentation about that subject in the case of a Win32 application. That's the reason why dealing with it is briefly explained in this part. Indeed, this may be a big gain of time for further users who may want to add dynamically changeable data via a Dialog Box (such as constraints on the movement and so on). For more information on Dialog Box, some links are given at the end of this report [15]. The msdn [10] library may as well be useful.

The dialog box is actually simple to build once the way to do it is explained.

✍ First make a new dialog box by choosing Insert, Resource, and then double clicking Dialog. An empty dialog box called Dialog1 appears, with an OK Button and a Cancel Button. It's possible to choose a different name for it.

✍ In the floating toolbar called Controls, it is possible to select and add edit boxes (places for the user to type strings or number as input to the program) and static text (used to label other controls such as edit boxes), and a lot of other kinds of buttons.

✍ It is subsequently necessary to create a new class calling "View", "Class Wizard …" from the menu. The name of the Class usually starts with C and contains the word Dialog. It is called CSizeDialog in the human model application.

✍ In the file SizeDialog.h (header file automatically generated when the Class Wizard made the class), it is necessary to include:

#include "resource1.h"

at the beginning of the code, 'resource1.h' (names like that in this application case) being a Microsoft Developer Studio generated include file containing the values of the ID.

✍ It's then possible using the Class Wizard dialog box to add member variables.

A member variable in the dialog box can be connected to a control's value or to the control. In the Class Wizard dialog box click the Member Variables tab. Then click on the IDC_NAME_OF_THE_CONTROL, and fill in a variable name as *m_name*, before selecting a type for the chosen variable (*CString*, *float*, …).

5.4.3 Making the Dialog Box work

The function *SaisieSize()* calling the dialog box code can now be written, since the dialog box is created.

First of all, dialog boxes are of two types: modal and modeless. The user must close a modal dialog box before the application continues or before going on other work. A modeless dialog box allows the user to display the dialog box and return to another task without cancelling or removing the dialog box, and then return to the dialog box. A call to the dialog object's *DoModal()* member function allows to display the dialog box and manages interaction with it until the user chooses OK or Cancel. This management by *DoModal ()* is what makes the dialog box modal. For modal dialog boxes, *DoModal ()* loads the dialog resource. The choice of using a modal dialog box seems to be a good one. Therefore, to display the dialog box, a call the *DoModal()* has to be done. *SaisieSize()* displays the dialog box and allows control of the variables. The code in the application fills the member variable of the dialog box with the body's member variables (Class Wizard added m_torsoh, m_torsow, … as public member variable of COptionsDialog, so the function can change them) and then brings up the dialog box by calling DoModal (). If the user clicks OK, the member variables change, and the human model is redrawn calling *Draw_Model ()*.

```
void CBodySize::SaisieSize(void)
{
	CSizeDialog sd;

	sd.m_torsoh=TORSO_HEIGHT;
	sd.m_torsow=TORSO_WIDTH;
	----The same is done for all the variables------

	int nResponse = sd.DoModal();

	if (nResponse == IDOK)
	{
		// Dismissed with OK
		TORSO_HEIGHT = sd.m_torsoh;
		TORSO_WIDTH = sd.m_torsow;
		----The same is done again for all the other variables------------

		// It is necessary to reinitialise Leg Height and Torso in here
		LEG_HEIGHT=UP_LEG_HEIGHT + LO_LEG_HEIGHT + FOOT_HEIGHT +
2*(FOOT_JOINT_SIZE+UP_LEG_JOINT_SIZE+LO_LEG_JOINT_SIZE);
		TORSO=TORSO_WIDTH / 3.0;
		Name = sd.m_name;
	}
	else if (nResponse == IDCANCEL)
	{
	// Dismissed with Cancel
	bHumanOn=TRUE; // ✍ The 'Change Size' item won't be checked  in the menu, since no changes
			//  have been done
	}
	Draw_Model();
}
```

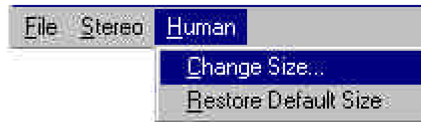
In order to compile this, it is necessary to add this line at the beginning of 'body.cpp':

```
#include "SizeDialog1.h"
```

This ensures that the compiler knows what a CSizeDialog class is when it compiles this file.

5.4.4 The application Dialog Box

As it is mentioned already, the size is dynamically changed calling a dialog box from the menu. Selecting "Change Size…" opens the Dialog Box with the default values in which the user can enter new parameters for the Human body, as well as a name for its model. It is therefore possible to define a model that matches anyone. Figure 34 below illustrates the display of the corresponding dialog box.



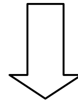


Figure 34. A model that matches anyone

5.5 Class behaviours for the application code

CMyApp (acts just as in the bottom application)**:**

✍ myApp:

Just creating this application object runs the whole application.

CMyApp myApp;

✍ InitInstance:

When any CMyApp object is created, this member function is automatically called. Any data may be set up at this point. Also, the main window of the application should be created and shown here. Return TRUE if the initialisation is successful.

CAnimAngles:

This class can be assimilated to a structure. It contains fifteen elements of type float, used to store the animation angles read from the file for each part of the body.

CBodySize:

✍ Human:

Creating an instance of this allows a human object to be created.

CBodySize Human;

✍ Draw_Model:

When a CBodySize object is created, this function allows all parts to be drawn together, and therefore to draw the complete model of a man (all the draw functions are declared in this class).

A call to *Human.Draw_Model()* to draw the model is done in the *OnPaint ()* function of the CMainWindow class.

✍ animate_body:

That's basically the animation function itself. It calls the *find_base_move ()* function also defined in this class.

✍ SaisieSize

It is the function called when the user selects the 'Change Size…' item from the menu, allowing the dialog box to be called and dynamically changing the human model size.

CSizeDIalog:

The ClassWizard automatically declares it. It essentially contains the variables used in the dialog box allowing the model size to be changed.

CMainWindow message map:

Associates messages with member functions (defines the member functions of the class). That's where the *OnPaint ()* function is called and performs the drawing.

A human model application now exists. It has many features (available via a menu) that are associated with the aim of this project, such as dynamic sizing or animation of the model. The user can choose to look at the model either in 3D or in the monitor plane. The model can also be animated or rotated. This actual model can still be improved in order to move the model in every wished direction, but Chapter 6 deals with that possible improvement, as well as others.

Chapter 6

6. Conclusion

This project is aimed at investigating the creation of 3-D virtual human models that can be used for image processing tasks. A student doing a Ph.D will continue this project over three years, the final objective being to train golf players. After a study allowing understanding how humans manage to represent the real world in three dimensions, it was therefore necessary to become familiar with 3-D graphics programming techniques (OpenGL, C++, Stereographics API), and to investigate the use of these techniques in human modelling. Implementing a 3-D visualisation application that demonstrates the validity of the human modelling approach taken, using both the investigated techniques and Stereographics hardware was the nest logical step of this project.

6.1 Summary

6.1.1 Work done

Thanks to their brain that processes what their eyes see, humans manage to create a 3-D representation of the world around them. This complex process is recreated in a computer environment by delivering separate left-eye and right-eye views in sequence, and using shuttering eyewear that is synchronised with the computer (these glasses alternately darken allowing each eye to see only the appropriate view). The drawing of the two views is done in OpenGL using 4 separate buffers. These consist of two sets of buffers: two back buffers (for the right and left eyes) into which the drawing is done, and two front buffers that are visible (on the monitor screen).

After having installed the graphics card and the required software for this project (Stereographics, Visual Studio,…) and read documentation related to the subject, doing tutorials appeared to be the best way to learn how to program in OpenGL. The choice to use OpenGL and not Java 3D was mainly due to the fact that OpenGL is a very powerful

graphics library (OpenGL provides great possibility to draw objects, animate or highlight them, but it needs some familiarity to use it).
Practicing tutorials [7] facilitated learning the basics (such as how to draw a window and add forms and volumes into it, how to animate these forms, add colours or light to them, etc…). Indeed, knowing the fundamentals is obviously essential to start the drawing of a real human model. An executable is now available for the further student continuing this project: it contains a pyramid and a cube rotating around two independent axes, with 3-D effects added to them. It's a working version that anyone can build on either to do their first steps or create their own application.

The following step was the creation of a bottom using the executable file created and the approach taken by Fotis Chatzinikos [17] in terms of human modelling. It implied to structure the model very carefully. The animation of the base according to the leg movement required a special relationship function, so that the model didn't seem to fly. A complete human model has then been built on this first model. Reading the animation angles in a file, and then rotating the corresponding limbs of the model causes the animation of the model. This has been possible due to the hierarchical structure of the model. The angles read corresponds to the key frames angles (at the moment there are twenty frames between two key frames when the application starts). The user can increase or decrease the number of frames (therefore increasing the precision in the case some are added, or on the contrary decreasing it in the case some are deleted). The angles between two key frames are calculated assuming that the movement is regular; If the hand goes from key frame A to key frame B with an angle of X, the rotation is done in twenty frames, with an angle of X/20 between two successive frames. The rotations are done in a single direction. Due to the duration of the project, no time was left to rotate the limbs in more directions, but an approach has been decided upon and is explained in detail later in this chapter. The model can also be rotated around its x-axis (using the menu), allowing a better view of the model and of its movement.

Other interesting features are provided by the application. The user can choose to have the stereo effects on (which implies to wearing the glasses) or not. In the case where the

stereo is on, the stereo parameters can be adapted to the user's sight. The human model can be rotated, animated, zoomed in or out (the zoom is only available without stereo, but could be added in the case there is stereo). Using the display window allows the model size to be dynamically changed. It permits a model that matches anyone to be defined, which is of great interest to train golf players. As an example if an ideal swing is recorded, it is be possible to map it back to the person.

To conclude with this project, tests have been done. They check the code as well as the visual aspect of the proposed model.

The final model is a good investigation to what can be achieved, and a good start for the following project. The animation is working well and has been explained very carefully. The main steps have been described (even how to start) and the major difficulties have been outlined and explained throughout this report in order to facilitate the work of a successor. In the next main paragraph, future directions or improvement that might be useful for further work in this area are also discussed (for instance an interesting way to improve the existing animation is proposed).

6.1.2 Problems encountered and solved

The first difficulty met was to rotate the quad and the pyramid of the executable along two different axes. The Pyramid was supposed to rotate on its Y-axis, from left to right, and the Quad on the X-axis, from back to front. Implementing this first gave an unexpected result, with a rotation added: the quad rotating around the pyramid Y-axis as on Figure 35.

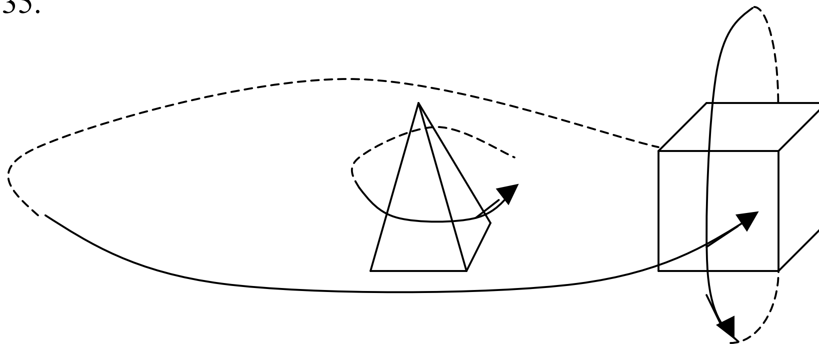


Figure 35. Non expected rotations

Indeed, while rotating objects or translating them, it is necessary to save the current configuration of the axes and to restore this configuration before doing any other transformation (in the case they are supposed to be independent). Indeed, while translating or rotating an object, the axes encounter the same transformations. This is illustrated on Figure 36.

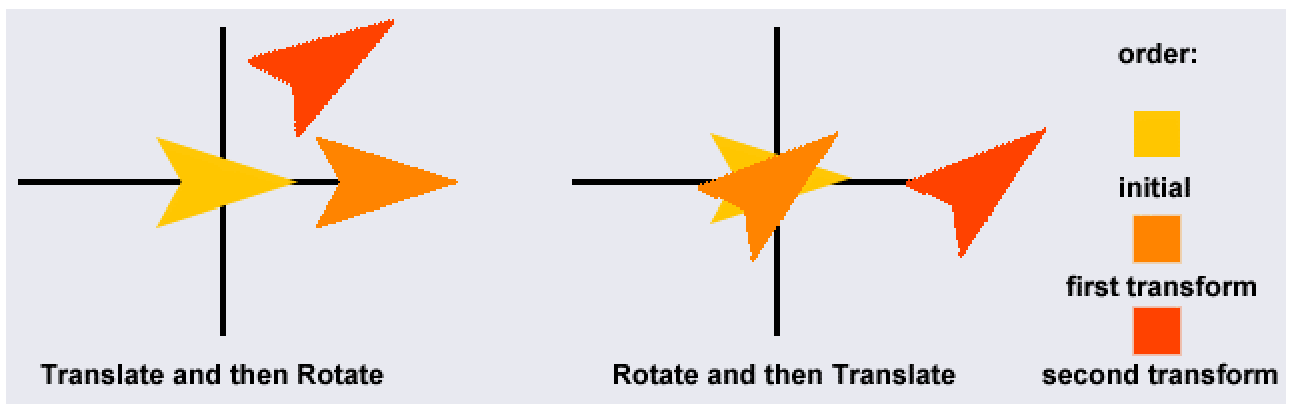


Figure 36. Transformation order

The original view can be restored by calling the *glLoadIdentity()* function. In the function *StereoProjection()*, a call to this function is made before doing the main transformation, so that every earlier transformation is reset, and it is not necessary to do it while calling it. An additional way of dealing with this problem is to make use of the *glPushMatrix()* and *glPopMatrix()* functions, as it is said in Chapter 3. These are used to draw the human model in a hierarchical way and to be able to rotate the limbs without rotating the whole body.

Another problem encountered was the installation of the glut library. The benefit associated with using this library is that it provides some more functions to draw the human model, such as *glutSolidSphere()* to draw a sphere or *glutSolidCube()* to draw a

cube (instead of drawing each face of the cube). The method to deal with it is provided in Appendix 1 (locations of the required libraries, options to be added into the project, etc…). It is also quite difficult to find out exactly what options need to be set while creating a new project. As an example errors were encountered because the "Debug Multithread" option was first not selected, but this can be avoid by reading Chapter 3, since it explains how to build a new project on which stereo is made available. How to make the stereo feature work was also found to be challenging and is explained in Chapter 2.

A major difficulty was to animate the body. The approach taken by Fotis Chatzinikos (C programming language, without stereo (3D) available) was very helpful. Anyway major changes had to be done so that the model animation can now be done in a straightforward maneer. Indeed, the explanation seems to be clear while reading its documentation, but the way limbs are moved in its code is not compatible. As an example, left leg angles make use of the right leg angles (which has no sense since the program is supposed to read the left leg angle and to move the left leg angle according to this particular value and not another one: otherwise there is no way to decide of the motion in a conscious way), or the animation angles are subtracted the ones to the others. The way animation is performed in the human model built for this project application is clearly explained in the Chapter 5.

The creation of a displayable dialog box called from the menu is not that easy as it may look when someone looks at the code. Indeed, a lot of documentation is made available in the case of an MFC application, while this project deals with a Win32 application. A lot of time has been dedicated to first of all learning how to deal with an MFC application, and then try to adapt this approach to the human model application. Once again, this is explained in detail in the fifth chapter.

Finally while using this dialog box to change the model size, it is imperative to record the total leg length, since that is used to calculate the base movement, otherwise the body jumps up an down while walking. Indeed, the function *find_base_move()* uses the total

leg length and compares it to the new distance between the feet and the top of the leg (by means of the new upper and lower leg sizes and the inclination angles).

6.1.3 Problems encountered but not solved

The different problems encountered in this part were not solved due to the short duration of the project, which is supposed to be an investigation and will be continued over three years.

Transparency of the objects appeared to be an unsolved problem. Indeed, having compact objects would look better: a body is not transparent, and seeing a limb that should be hidden might be felt as a visual nuisance. But due to a lack of time, more effort has been spent on more important things, with thought that a further user might solve it.

The zoom feature is only available if no stereo is on, since zooming with the 'stereo on' requires recalculation of all the parameters of the stereo projection. Indeed, translating along the z axis after having called the *StereoProjection()* function (in the *OnPaint()* function) brings a strong feeling of discomfort for the view if the parameters are not readapted, and that appeared to be quite hard. It is anyway possible to adapt the stereo parameters using the menu, but it would be much better to include the modification while zooming, that's why no zooming is actually made available only while the stereo is on.

And apart from these unsolved problems the future guidelines that might be taken to continue this work (and that would have been taken if more time had been available) are now discussed. These are discussed in the following paragraph.

6.2 Future Work

In this section the future directions that the student continuing this project might usefully take are presented.

6.2.1 Geometrical improvement

First of all the default size could be changed using a more precise model. But that's a minor improvement since the size is dynamically changeable to match anyone. Indeed, a more interesting aspect would be to save in a file the new model that is entered and be able to use it afterwards whenever it is necessary, without needing to re-enter the sizes. Just writing the sizes in a file (named using the new model name) and reading them whenever it is necessary might achieve this. As an example, when entering a name in the dialog box, the dialog box may test that name. If the name already exists, the dialog box may open and read in the corresponding file the sizes of this particular model and automatically reuse them.

Always with the aim to have a more accurate model, adding joints might be a point of interest. Indeed, another joint at the top of the torso (corresponding to the chest) and joints on the feet (Figure 37) would be a good and straightforward improvement to the existing model, allowing more degrees of freedom in the animation.

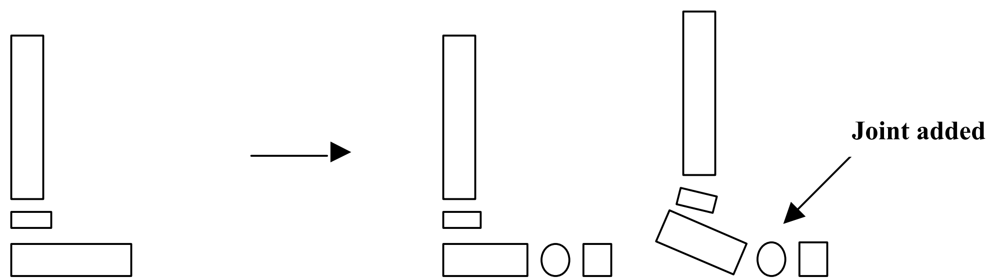


Figure 37. A: Actual foot, B: Possible improved foot

Another interesting aspect would be to add skin to the model to have a more human looking and a more elaborated model. A possible technique [17] would be to analyse some good two-dimensional photographs of a human body and try to retrieve the underlying three-dimensional model. Once an appropriate number of points are found (x, y and z coordinates according to axes assigned to the human photograph), it is then possible to create a three dimensional data set of a human model that can be used to

create the OpenGL basic model. Once the points are available to the program, instead of drawing a cube for each limb of the body, some functions can be built and used for each one of them, using the appropriate set of data. All the functions for creating the body parts would be very similar. Such a function would draw the key points (to draw the outline of the limb, while using its symmetry to facilitate the work). To create the actual surface it is then just necessary to connect these points. A variety of connections is conceivable. Using GL_TRIANGLE_STRIP values is one possibility. It connects points as shown on the four top points of the Figure 38 below (this figure represents the front of the torso).

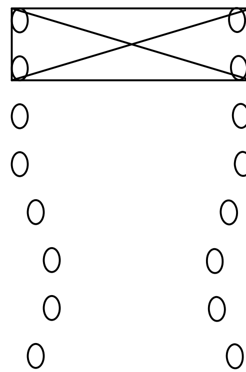


Figure 38.Creating triangle strips

The function drawing the torso may then be built in the following way, taking advantage of the symmetry of the human body.

```
Create_torso(data_set_of_the_points_used_to_create_the_torso)
{
        create_torso_front(torso[front][left], torso[front][right]);
        create_torso_back(torso[back][left], torso[back][right]);
    create_torso_sides(torso[front][left], torso[front][right], torso[back][left], torso[back][right]);
}
```

The work of Bruno Heideberg [16] to create a skin is very impressive (as well as his whole work by the way). The skin was in that case generated in a different fashion from the one described above. It might be harder, but the results are very good. The way it is done might be by using millions of polygons, and his application makes use of 3Dstudio max. The resulting human model can be seen in Figure 39. The use of light obviously increases the quality of the visual aspect.

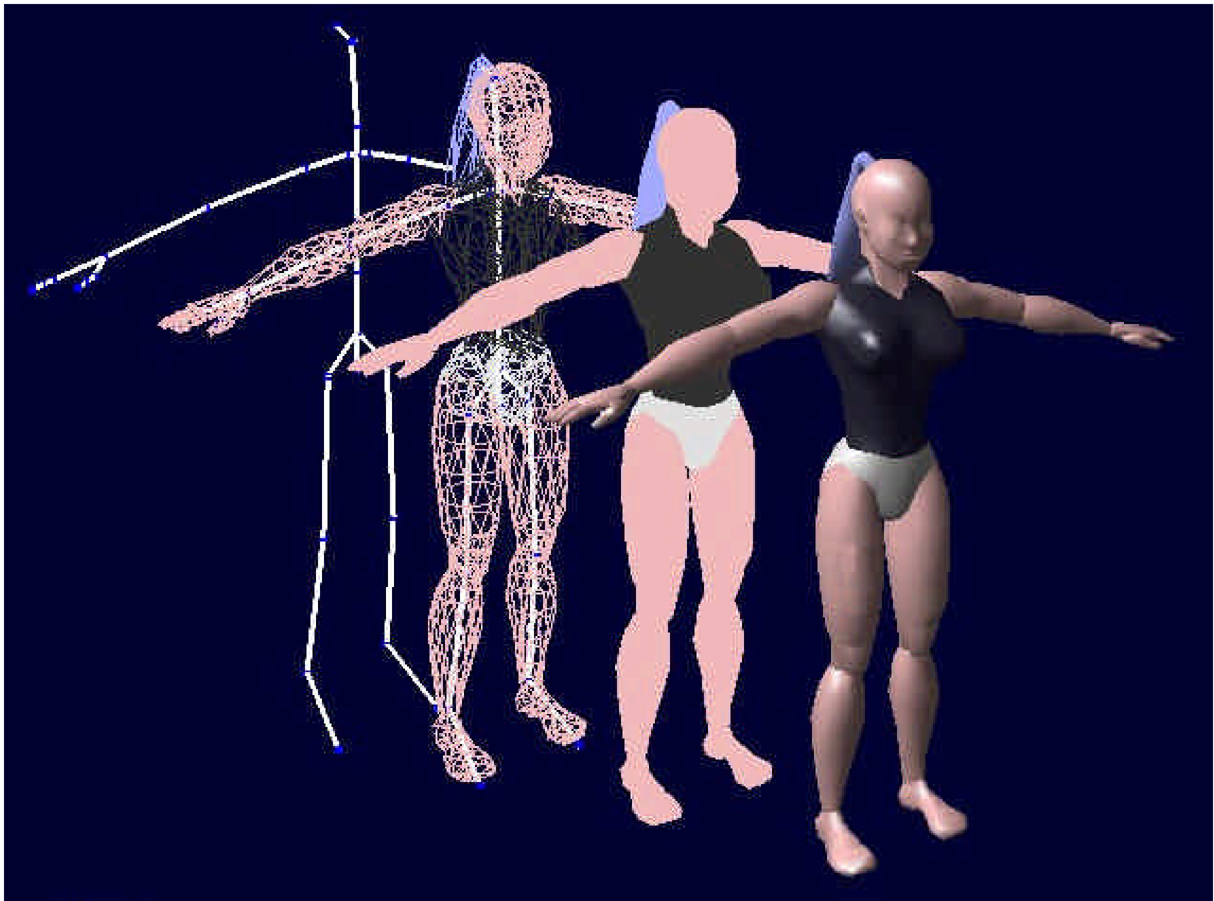


Figure 39. Cal 3d demo

6.2.2 Motion control

The actual human model is only able to perform symmetrical movements since in the implementation the key frames are assumed to be symmetrical (the last four to the first four). Indeed, the last four key frames make use of the angle values of the first four ones

(since 8 key frames are used, the case (5) in the switch loop of the *animate_body ()* function corresponds to the symmetrical version of case (1)).

To allow non-symmetrical animation, the angle values of the eight key frames have to be entered into the data file. That would allow reading the angles for the four last key frames directly from the file, and the movement would not necessarily be symmetrical any more. This implies that the *angles* array used to store the angles has to be increased in size (it might be dynamically allocated, allowing the size to be increased in an easier way): *angles[7]* instead of *angles[3]*. Then the *animate_body()* function would need to be changed to use these new angles instead of the ones of the first four key frames for the second half of the animation, but that's very easy to change. Anyway this way of thinking implies a periodic movement, since it is still repeated every 8 key frames. In the case of a swing, that's not a problem, since it's one movement, always supposed to be the same. The human model may just be programmed to do it once.

Another interesting aspect may be to add some constraints to the model. At the moment, it is necessary to think carefully before entering the angles in the 'data.txt' file, since the model uses these angles even if they are not realistic. A way to add constraints that matches anyone may be done using a dialog box called from the menu, similar to the one used to size the model. Indeed, a human can't bring its arm too far in the back (Figure 40), further from a certain angle (this angle depending on the person even if it's still in the same direction). It might therefore be interesting to define a set of constraints (values that could be changed dynamically) in each part of the body and test the entered angles according to these constraints.

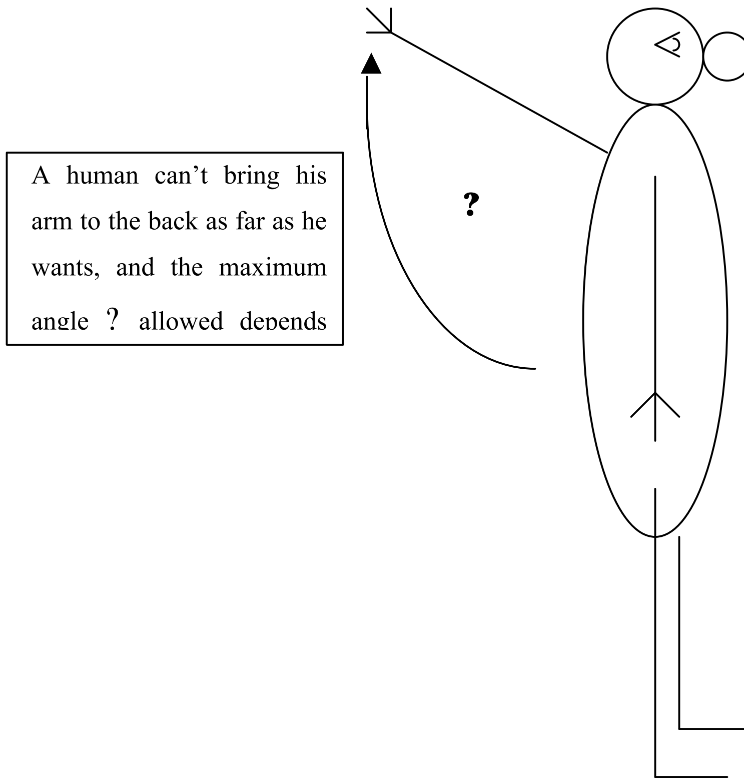


Figure 40. Constraints exists in the body

Since no constraints exist in the model at the moment, motion capture appears to be the clever way to improve the existing geometric model. Among the best-known and most direct methods for animating skeletons, motion capture may be considered. It consists of the detection of a human body movement. Using either sensors (to provide coordinates of specific points of joint angles of a real human for each frame captors positioned on the articulations (this is called rotoscopy)) or a system of visual markers recorded by cameras allows doing this. These methods are explained in more detail in the second chapter of this report. It actually involves measuring a human's position and orientation in physical space, and then recording that information in a computer-usable form. Once that is done, animators can use the recorded data to control elements in a computer-generated scene. Attractive aspects of this method are its rapidity and the fact that it does not require specific talents to get good quality information. Usually all the required information to the human model animation are not measurable; therefore a certain amount of key positions are measured, and the remaining angles are calculated using techniques such as

the inverse kinematics. In the case of the existing application, a golf player may execute a perfect movement, well calibrated, and this movement may be recorded for good. The angles of each joint would then be extracted and saved. They may then be passed to the file 'data.txt' containing the animation angles of the human model. The information recorded in this manner may then be used for every subsequent user. Actually, the model has to be improved in order to match this application, since the joints can only be bent in a single way, while in the real world some of the joints (like the wrist) can be bent in at least two directions. Since motion capture allows real time animation of the model, and directly from the user body, it may then be possible to compare the subsequent user angles gained to the "perfect" recorded ones. The perfect swing recorded may also be visualized and compared to the user's one.

Anyway in order to be able to use motion capture, it would be more interesting to have movement allowed in the three dimensions. Building on the existing model can be achieved by adding two dimensions to the *walking_angles* array (and the corresponding values for the angles to be read into the data field). It would be similar to a vector in a sense. Instead of having a walking vector like the actual one,

walking_angles[side][x]

the array would have memory allocated for the y and z direction as well :

walking_angles[side][3][value],

with in second position the meaning of the value being:

0 for the x axis,

1 for the y axis,

2 for the z axis,

The parameter *value* would correspond to the amount of rotation along the given axis. Indeed, the rotation of the member is actually done while drawing the limbs, and using the *glRotatef(angle_value, x_direction, y_direction, z_direction)* function. It would be possible to do the rotation along the x axis, then along the y axis, and to end up along the z axis, with care to save and restore the original references before each rotation. It is therefore possible to bring the movement into three directions and implement this without too much difficulty. To reach the same objective of movement in the three directions, it

might otherwise be interesting to have a different approach and think of the movement in term of vectors as defined below. It would allow changing the motion in the three directions according to two rotation vectors and translations or relations between the limbs (similar to the relation existing between the base and the legs of the body: if the legs are bent the base goes down without any need to define a translation vector).
The first vector may be in the Y direction (Figure 41, A) and it may represent the rotation around the Y-axis. The size of the rotation may be related to its speed (the longer the vector is, the faster the rotation is) and the sign to its direction (right to left or left to right, in other terms positives to negatives or negatives to positives). The second vector would then be along the X-axis (Figure 41, B) and would represent the rotation around this axis, once again with the sign of the vector indicating the direction of the rotation (back to front or front to back), and the magnitude of the vector its speed. The use of these two vectors would allow movement in the three directions. The vectors could then be added to add different motions together.

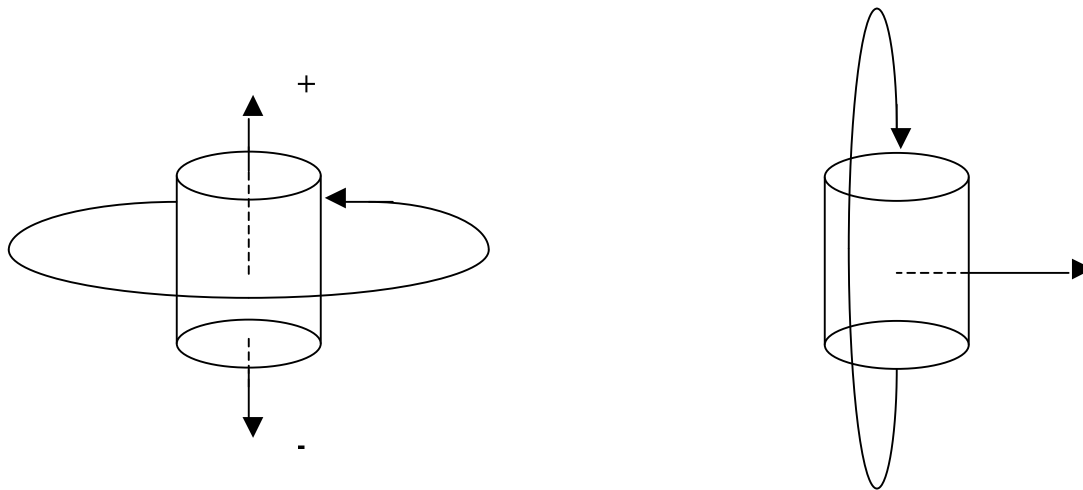


Figure 41. A: Rotation along the Y-axis, B: Rotation along the X-axis

This kind of motion should be used at the joints. Anyway it's just a theory, and the way to implement this method is still to be finalised.

It has been seen in this Chapter that the work has been completed, allowing a good start for the Ph.D student. A human model with all the desired features has been developed. The model is built in C++ using the OpenGL graphics library (Java3D could also have been used, but OpenGL has been chosen because it is a very powerful library). The model was coded step by step and an executable is made available to help the Ph.D. student that will then continue this project. Indeed it is possible for the student to build on this code to help with their understanding of 3-D graphics programming techniques.

A menu allows the user to view the model in 3-D, to resize it or to restore its default size, to rotate it at different speeds, and to animate it. A zoom feature is also made available in the case where stereo is off. If the user decides to look at the model in 3-D, they can change different parameters via the menu, adapting the 3-D effects to their sight (See chapter 2 section 2.4.1), and therefore having a more comfortable look at the model. Further investigations have also been performed, providing further directions that might prove interesting and useful.

References

Stereographics Corporation Web Site:

1. http://www.stereographics.com/

2. StereoGraphics Developers' Handbook

Stereoscopic and OpenGL:

3. Duoduo Liao, Shiaofen Fang, "The design and application of high-resolution 3D stereoscopic graphics display on PC"
   Department of Computer & Information Science
   723 W. Michigan St, SL280
   Indianapolis, IN 46202, U.S.A.
   http://wscg.zcu.cz/wscg2000/Papers_2000/R5.pdf

4. R. Rodriguez, G. Chinea, N. Lopez, "Full Window Stereo"
   Centre for Genetic Engineering and Biotechnology, Havana, Cuba.
   G. Vriend.
   BIOcomputing, EMBL, Heidelberg, Germany

5. http://www.info.ucl.ac.be/etudiants/Cours/2760/ex1/2760ex1.html

OpenGL:

There are quite a few sites with information on OpenGL but the most useful one proved to be the one corresponding to the reference number 6.

6. http://www.opengl.org/

7. Jeff Molofee's OpenGL Windows Tutorials
http://nehe.gamedev.net/opengl.asp

8. OpenGL Reference Manual (Addison-Wesley Publishing Company)
http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI_Developer/OpenGL_RM/@Generic__BookView;cs=fullhtml;pt=4?DwebQuery=glLightfv

9. OpenGL Programming Guide (Addison-Wesley Publishing Company)
http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI_Developer/OpenGL_PG

10. Index of the OpenGL commands
http://www.cevis.uni-bremen.de/~uwe/opengl/opengl.html

MSDN Online search:

11. http://search.microsoft.com/us/dev/default.asp

C++ and Visual C++:

12. Christian Casteyde, 2001. "Cours de C/C++"
http://www.developpez.com/c/megacours/book1.html

13. Brian "Beej" Hall, "Beej's Guide to Network Programming"
http://www.bitafterbit.com/english/c/networking/index.html

14. Jacques E. Zoo,
Visual C++ and MFC tutorials using Visual C++6.
http://www.functionx.com/visualc/

15. "Special Edition Using Visual C++",

Chapter 2 “Dialogs and Control”
http://nps.vnet.ee/ftp/Docs/C/Using%20Visual%20C++%206/ch02/ch02.htm

Human and motion:

16. Bruno ‘Beosil’ Heideberg, 18 May 2001.
http://cal3d.sourceforge.net/

17. Fotis Chatzinikos, “A 3D case Study using OpenGL”
http://www.dev-gallery.com/programming/opengl/book/

18. Tim Naylor, 2000. “Character Setup for Real-Time in Maya 3.0”
http://www.highend3d.com/maya/tutorials/realtimechar1/index.3d

19. “Animation de l’avatar”
http://dept-info.labri.u-bordeaux.fr/~parisy/ExposeRV/AnimationAvatars.html

20. “The Virtual Football Trainer”,
University of Michigan,
Virtual reality Laboratory at the College of Engineering.
http://www-vrl.umich.edu/project/football/

21. Skill Technologies, Inc.
3D-Golf Viewer Lite Version 1.0
http://www.skilltechnologies.com/Skilltec/3GDVover.htm

22. Daniel Thalmann, 1996. “Physical, Behavioural and sensor-based Animation”, Computer Graphics Lab, Swiss Federal Institute of technology.

23. Scott Dyer, Jeff Martin and John Zulauf, 1995. “Motion Capture White Paper”.

24. Gael Sannier(1), Selim Balcisoy(2), Nadia Magnenat Thalmann(1), Daniel Thalmann(2), 2000. "VHD: a system for directing real-time virtual actors".

1) MIRALab, University of Geneva, 24 rue du general Dufour, CH 1211 Geneva, Switzerland.
2) Computer Graphics Laboratory, Swiss Federal Institute of Technology, EPFL, LIG, CH 1015 Lausanne, Switzerland.

25. Deepak Tolani, Ambarish Goswami, Norman I.Badler. "Real-time inverse kinematics techniques for anthropomorphic limbs".
University of Pennsylvania, Computer and Information Science department, Philadelphia, PA 19104-6389

Appendix 1: How to install the glut library

I got this help from the following site:

http://www.sdsc.edu/~mjb/ames293/gettingglut.html

?? Click here to get glut.h

Install this file as: **C:\Program Files\Microsoft Visual**

Click here to get glut32.lib

Install this file as: **C:\Program Files\Microsoft Visual Studio\VC98\Lib\glut32.lib**

Click here to get glut32.dll

Install this file as: **C:\WINNT\system32\glut32.dll** if you are on Windows NT, or in: **C:\WINDOWS\system32\glut32.dll** if you are on Windows 95 or 98.