2008

Mocap Viewer

CM20219 Fundamentals of Computer Graphics

This is the documentation for the CM20219 Graphics coursework. The handout supplies information on the techniques used and problems faced during the implementation of a Motion Capture Viewer. It also provides details of the Mathematics used in OpenGL.

Benjamin Bourdin
bb247
12/9/2008

Documentation

Table of Contents

Introduction 3

Implementation of requirements in OpengGL: 3

1.Code must display skeleton in initial pose: 3

Drawing the joints: 3

Drawing the bones: 5

Spherical Coordinates: 5

Aligning Z-axis: 5

Drawing the cylinder: 6

Use of the 'f' key: 7

2.Code must display skeleton moving under mocap data: 8

Euler angles: 8

Introduction of a frame counter: 9

Translate and rotate by root_pos and root_orient, RrTr: 9

Matrix chain for animation K'TRK: 10

Rotate into the arbitrary axis, K: 10

Rotate by bone orientation, R: 10

Translate by bone direction, T: 10

Rotate out of arbitrary axis, K': 11

3.Code must allow interactive specification of camera viewpoint: 12

Camera coordinates: 12

Keyboard Interaction: 13

Orientation of reference frame: 13

Problems faced: 14

4.Include simple visual aids – e.g. Texture mapped floor, or Reference Frames: 17

Loading Texture: 17

Image texture: 17

Draw reference frames: 19

5.Camera must track movement of skeleton: 19

Calculate root position: 20

Calculate camera position: 21

Introduction

This is the documentation for the CM20219 coursework. It consists of designing a motion capture viewer for asf and amc files. In this document I will go through each requirement and explain how I implemented it and what mathematics or techniques I used. I only implemented requirements one to five giving me a maximum of 105 marks. I found this coursework interesting and challenging at the same time. It also gave me a much better understanding of graphics in OpenGL and 3D in general.

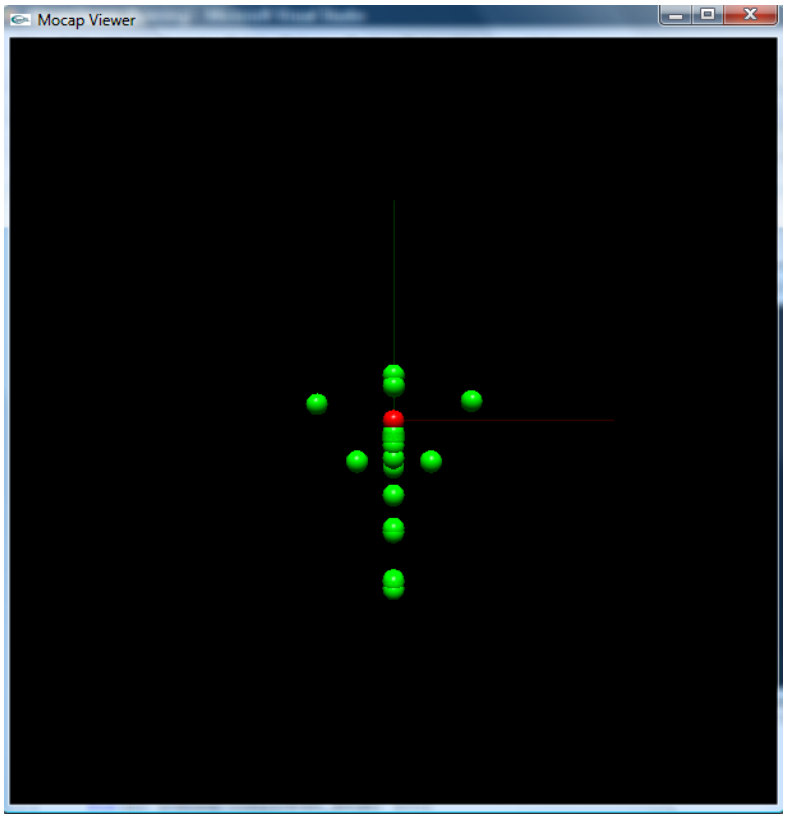
Implementation of requirements in OpengGL:

1. Code must display skeleton in initial pose:

Drawing the joints:

Once I had the initial code working, displaying the teapot, the first thing I did was look at the moodle forum for information on how to start. The discussion "glTranslate / drawing the skeleton" gave a very useful hint on copying the recursive structure of `parser_debugskeletonTree(...)` and `debugskeletonTree_recur(...)`.

So I used these functions and copied their recursive structure in **drawSkeleton(...)** and **drawJoints(...)**. The functions have the same structure but instead of printing spaces and then the name of the bone I translated by the bone->direction*bone->length (T), drew a sphere, and then applied the inverse translation (T'):

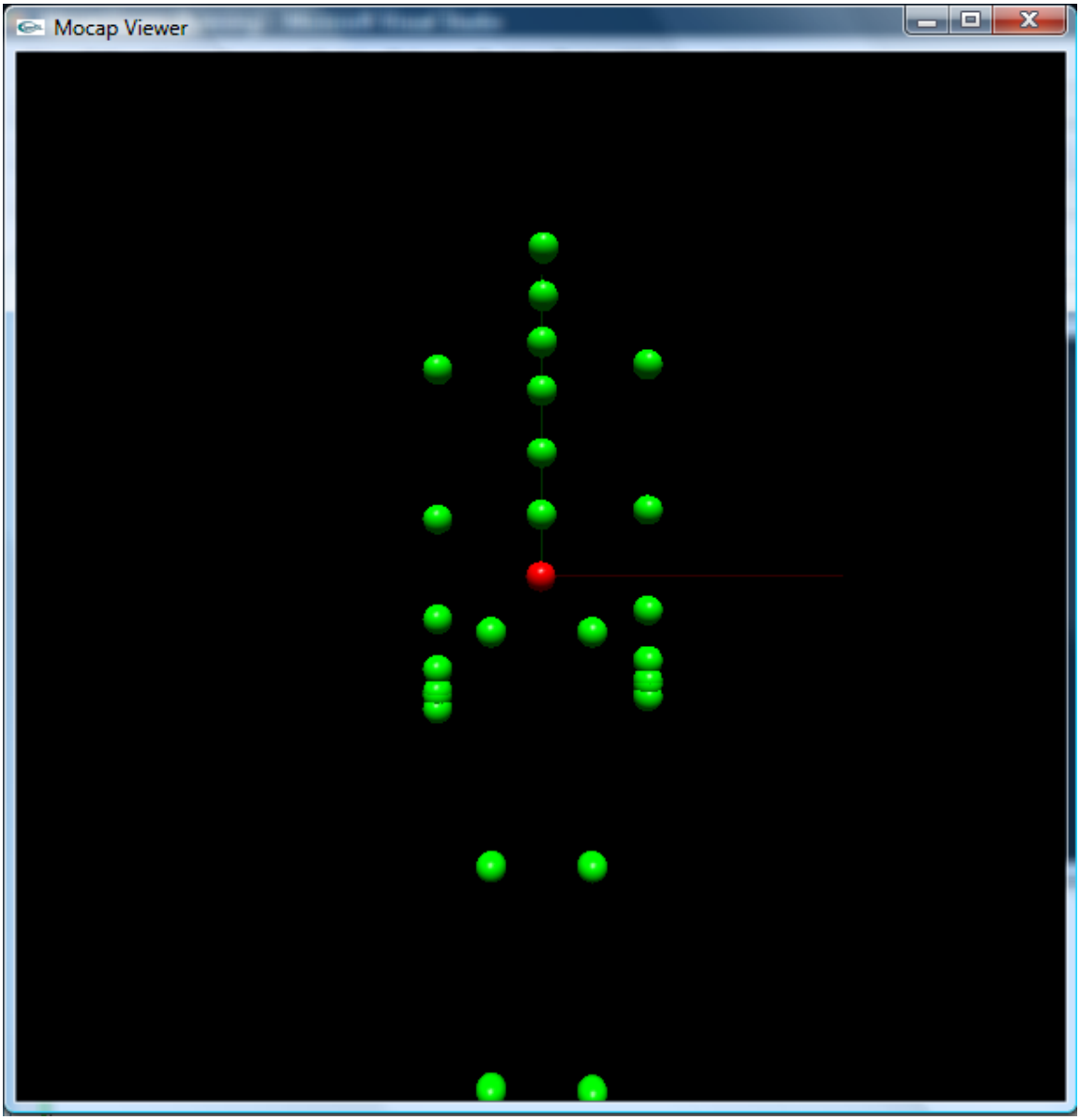


The problem here is that after having translated by the bone direction the reference frame is brought back to the root (original reference frame). So the next joint will be drawn from the root reference frame. Since the direction vector, specified in the bone struct, is defined from the parent joint and not the root, the skeleton joints will all appear placed near the root.

In order to fix this problem the inverse translation needs to be made after having drawn all of the joints children, so that the children "inherit" the initial translation(s) from the root to the parent. The children are drawn from their parent reference frame. Code is replaced by:

```
glTranslatef(x, y, z);

glutSolidSphere(...);

for(all children of bone){ drawJoints(...); }

glTranslatef(-x, -y, -z);
```

As a result the skeleton joints are drawn correctly:

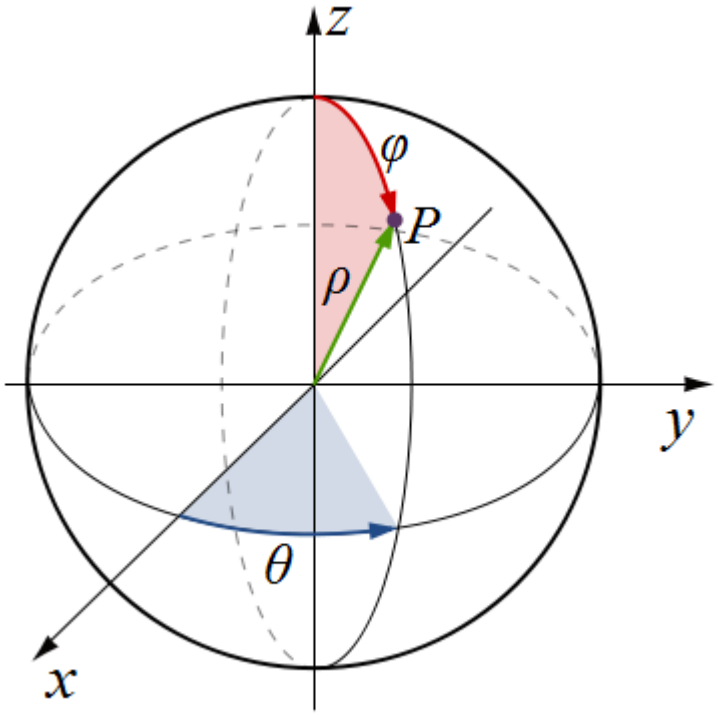


Drawing the bones:

It is obvious that, looking at the reference implementation, the bones are drawn using cylinders. The function for drawing cylinders is **gluCylinder(...)**. This function requires the use of a **GLUquadric*qobj** and it draws the cylinder on the Z-axis. So I created a new function called **drawCylinder()** which creates a new GLUquadric object, rotates the reference frame so that the Z-axis is pointing towards the next joint and then draws the cylinder.

Spherical Coordinates:

Spherical coordinates are polar coordinates in 3D. Instead of having only one angle "theta" between the X-axis and Y-axis, it has another angle "phi" between the Z-axis and R. The following diagram illustrates:



$$x = \rho \sin\varphi \cos\theta$$
$$y = \rho \sin\varphi \sin\theta$$
$$z = \rho \cos\varphi.$$

$$\rho = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = \operatorname{atan2}(y, x),$$

$$\varphi = \operatorname{atan2}(\sqrt{x^2 + y^2}, z) = \arccos\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right)$$

P is of coordinate (x,y,z). We need to align the Z-axis with the vector P, showed in green in this diagram.

Aligning Z-axis:

The alignment of the Z-axis with the point is just going to be a series of rotations. The first one is a rotation of 'theta' about Z, so X is in the ZP plane (plane containing the point P and Z-axis)

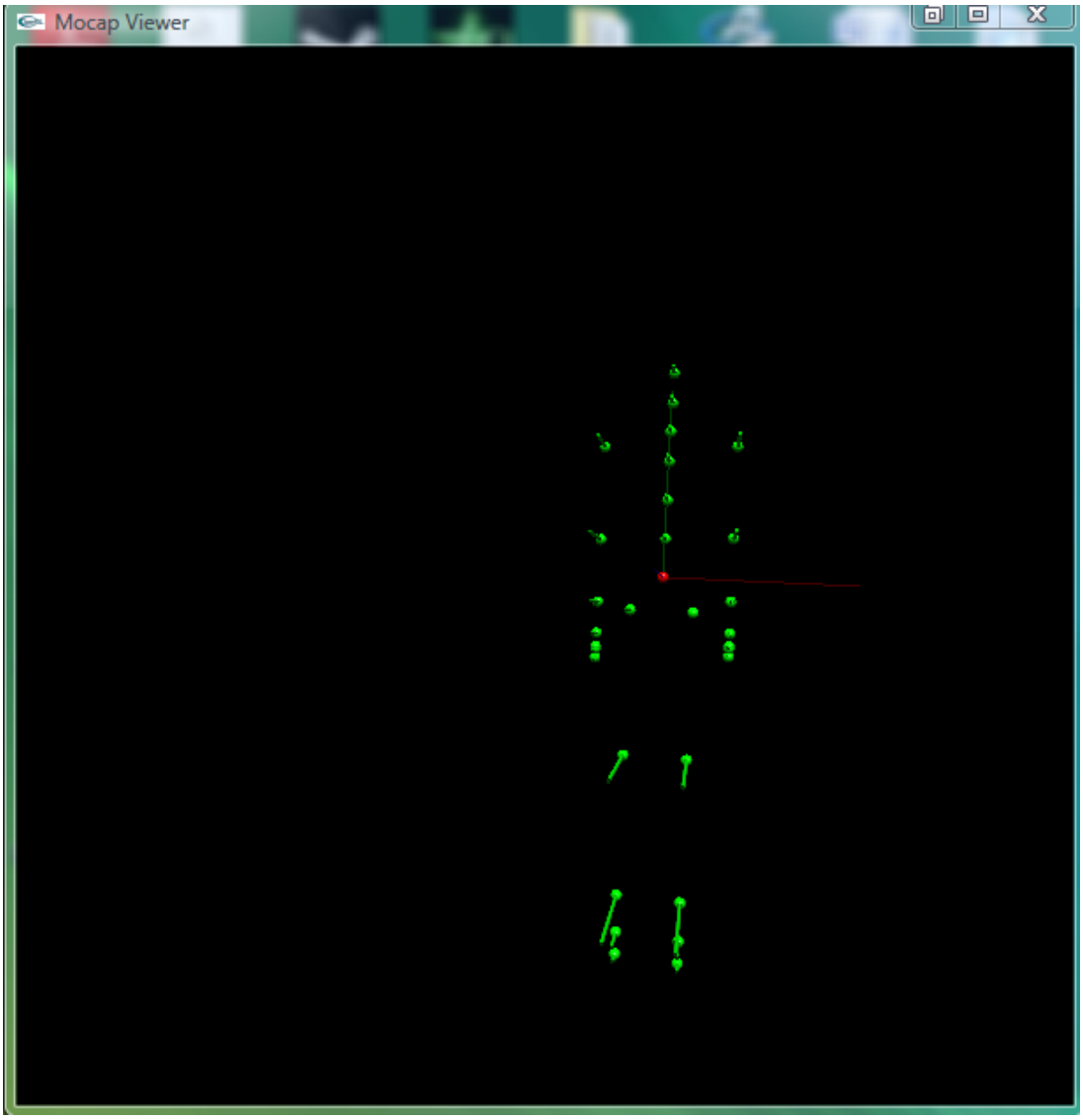
and Y is the normal to that plane. We are then going to rotate of 'phi' about the Y-axis so Z comes into line with P.

Drawing the cylinder:

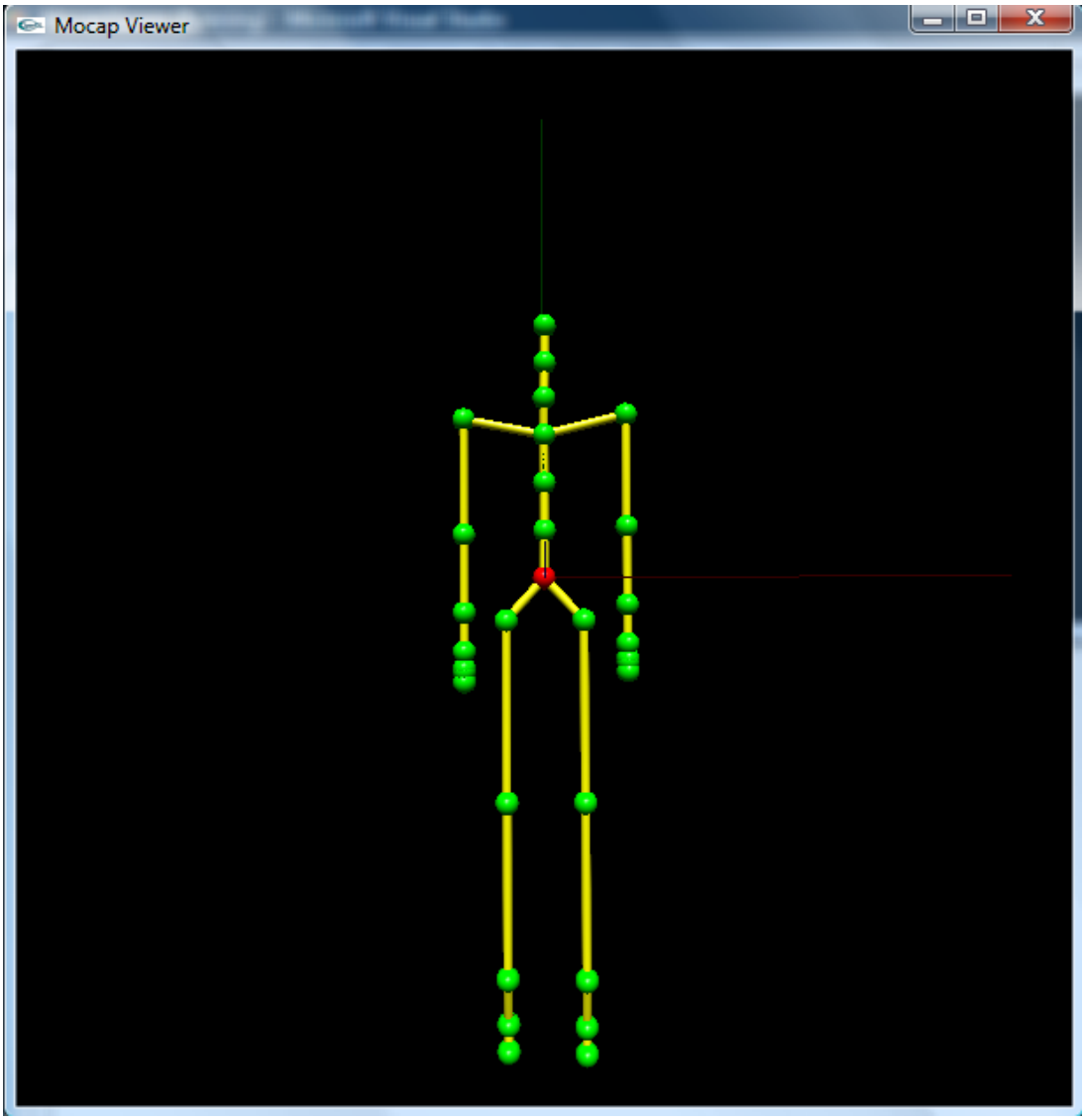
We can now draw the cylinder with **gluCylinder(qobj, baseRadius, topRadius, height, slices, stacks)**. We are using the same number of slices and stacks as the spheres, which is defined as **SLICES** and **STACKS** in my code. The **baseRadius** and **topRadius** are going to be the same otherwise we would be drawing a cone like shape. The height of the cylinder is simply going to be equal to R which is the length of the bone. Step by step the code is going to look something like this:

```
Calculate angles

Rotate of theta about Z

Rotate of phi about Y

Draw the cylinder
```

This yields:



The skeleton joints are drawn correctly but the bones appear to be drawn around the Z-axis without it being rotated. This is actually due to the **atan2()** and **acos()** functions of the math library. They return radians whereas the **glRotatef()** function in OpenGL takes degrees as a parameter. All we have to do now is convert 'theta' and 'phi' into degrees before rotating, which gives us:



The skeleton and bones are draw correctly, and the initial pose is finished.

Use of the 'f' key:

The user should be able to toggle between initial pose and mocap data by pressing the 'f' key. The way to do this is to introduce a Boolean as global variable so when the user presses 'f' the Boolean is set to true or false depending on its previous value. If it was false set it to true, if it was true set it to false. We also need to create an 'if' statement in our display function. If the Boolean is true then the 'f' key was pressed and display the skeleton in its initial pose, or else display it under mocap data. We therefore have two different **drawSkeleton** and **drawJoints** functions: **drawInitialPose** and **drawInitialJoints**. This concept of "toggle" will also be used for the reference frames. The user can toggle on or off the reference frames by pressing 'r'.

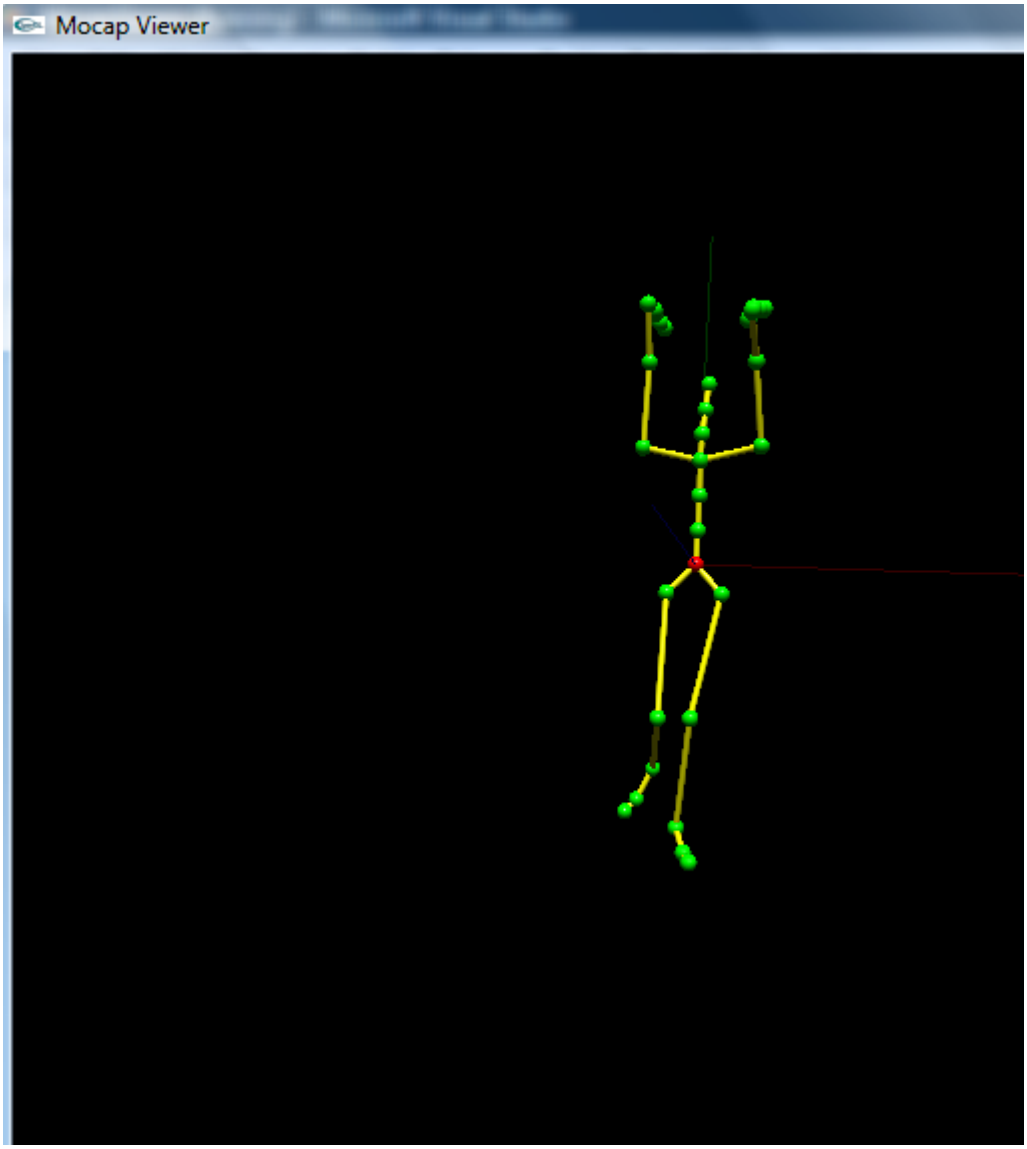
2. Code must display skeleton moving under mocap data:

To animate the skeleton we need to apply the matrix chain described in the lectures: K'TRK RrTr to the identity matrix.

Euler angles:

First of all we need to describe Euler angles and understand how rotations work in OpenGL. Euler angles are a means to describe the orientation in space of any frame as a composition of rotations from a reference frame. There are three such matrix rotations: Rx, Ry and Rz which rotate about the X-axis, Y-axis and Z-axis respectively. A rotation using Euler angles is performed using some sequence of Rx, Ry and Rz. The order of multiplication does matter as some orderings can lead to a Gimbal lock. This is why we will use the order RxRyRz in our rotations.

For example here I used the order RzRyRx and the arms of the skeleton are orientated the wrong way:



In OpenGL rotation are used by calling the **glRotatef(angle, x, y, z)**. The Rx rotation matrix for example will be in OpenGL: **glRotatef(theta, 1, 0 ,0)**. The (x, y, z) in **glRotate** specify which axis we want to rotate about. Euler angles are defined by three angles x, y, and z for Rx, Ry and Rz. So Rx*Ry*Rz will be in OpenGL:

```
glRotatef(z, 0, 0, 1); glRotatef(y, 0, 1, 0); glRotatef(x, 1, 0, 0);
```

Introduction of a frame counter:

We have to establish a counter that iterates between all the frames. The counter is initialised at 0 and is incremented until it reaches the number of frames of the mocap data. The code is similar to the one in the teapot orbiting sphere demo on moodle. We are going to use the **idle()** callback to update the state of the animation, here that is the frame counter. So in the **idle** function we increment the frame counter and if the frame counter is bigger than the number of frames then we bring it back to 0.

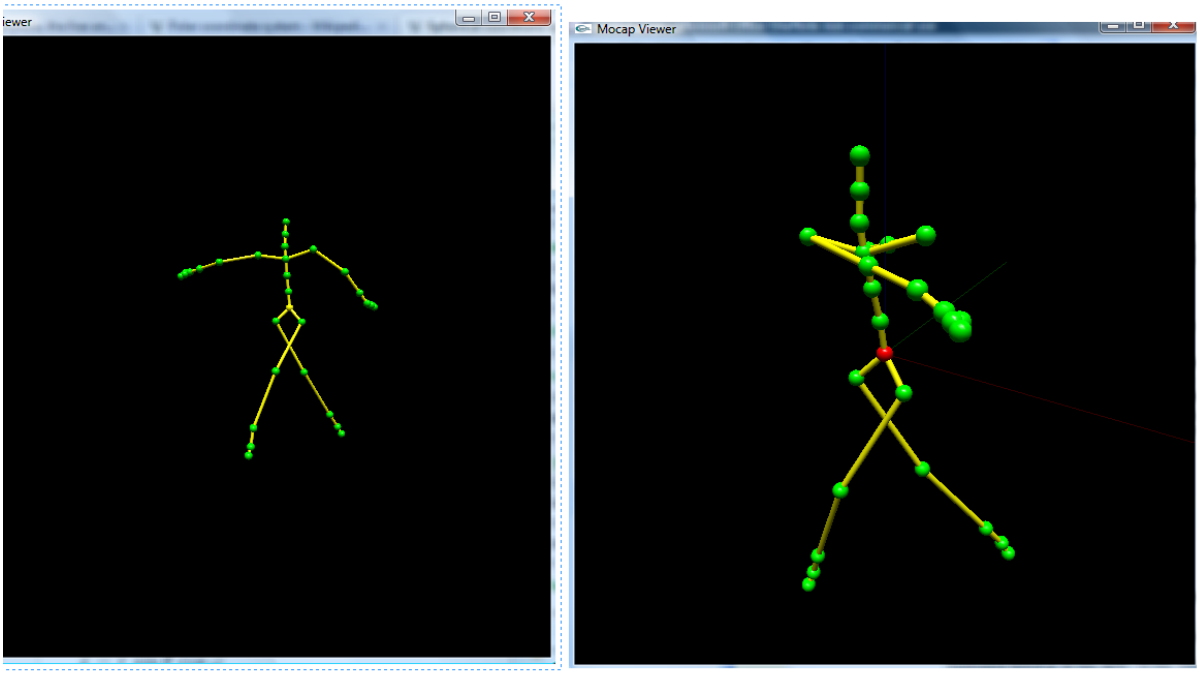
Translate and rotate by root_pos and root_orient, RrTr:

Right before drawing the root we have to translate and rotate by the **root_pos** and **root_**orient, respectively, of the current frame. This is the RrTr in our matrix chain which is implemented as follows:

```
glTranslatef(gMo->root_pos[frame].x, gMo->root_pos[frame].y, gMo-
>root_pos[frame].z);


glRotatef(gMo->root_orient[frame].z, 0, 0, 1);
glRotatef(gMo->root_orient[frame].y, 0, 1, 0);
glRotatef(gMo->root_orient[frame].x, 1, 0 ,0);
```

This will make sure the root is actually moving as in the reference implementation. Of course after having drawn the skeleton we have to take out this translation and rotation. We want to restore the initial identity matrix after having drawn the skeleton. To do that we save the identity matrix, using **PushMatrix()**, at the very beginning of the **drawSkeleton()** function and then we load the saved matrix with **PopMatrix()** after having drawn the skeleton with all its transformations.

Currently we have the skeleton in its initial pose moving around the scene. We need to introduce a rotation before the translation of the bone in **drawJoints()** to animate it. For help I looked at the "glTranslate / drawing the skeleton" discussion on the moodle forum and the lecture notes on the matrix chain used for a 3D rigid body transformation. I first implemented the skeleton movement without the rotation into the arbitrary axis K. The skeleton's movement were incorrect:



In these screenshots seen from different points of view the K rotation is not included in the matrix chain.

In these screenshots seen from different points of view the K rotation is not included in the matrix chain. We now have to apply the matrix chain K'TRK to the identity matrix to draw the skeleton appropriately.

Matrix chain for animation K'TRK:

Rotate into the arbitrary axis, K:

The arbitrary axis angles for each bone are stored in **bone->axis**. So we do:

```
glRotatef(bone->axis.z, 0, 0, 1);
glRotatef(bone->axis.y, 0, 1, 0);
glRotatef(bone->axis.x, 1, 0, 0);
```

Rotate by bone orientation, R:

The orientation of the bone for the current frame is stored in

gMo->bones_orient[frame][bone->id] . So the code is:

```
glRotatef(gMo->bones_orient[frame][bone->id].z, 0, 0, 1);
glRotatef(gMo->bones_orient[frame][bone->id].y, 0, 1, 0);
glRotatef(gMo->bones_orient[frame][bone->id].x, 1, 0, 0);
```

Translate by bone direction, T:

Just as for the initial pose we apply the translation:

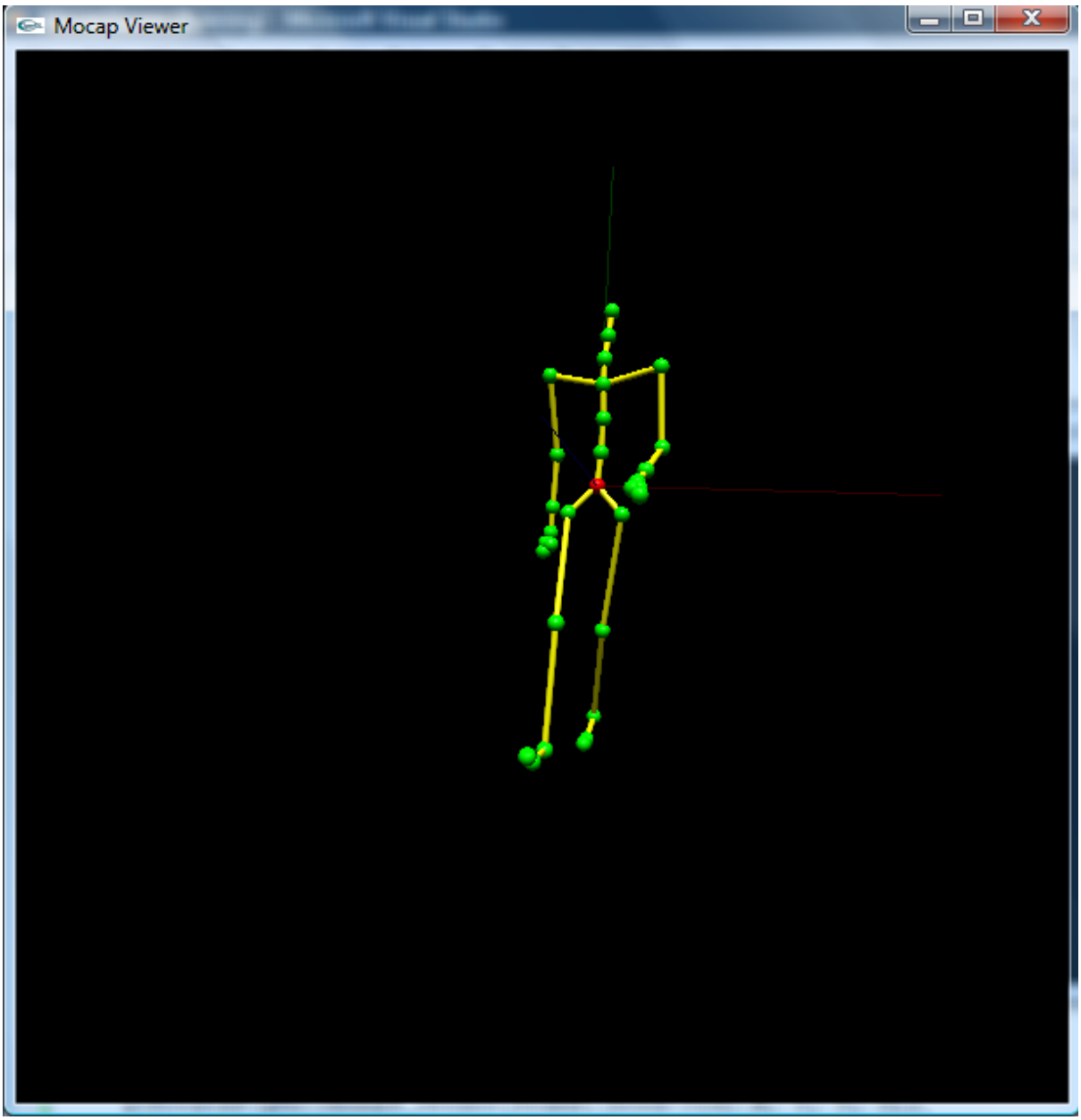
```
glTranslatef(bone->direction.x*bone->length, bone->direction.y*bone-
>length, bone->direction.z*bone->length);
```

Rotate out of arbitrary axis, K':

We just apply the opposite of K:

```
glRotatef(-bone->axis.x, 1, 0, 0);
glRotatef(-bone->axis.y, 0, 1, 0);
glRotatef(-bone->axis.z, 0, 0, 1);
```

The result is as follows:

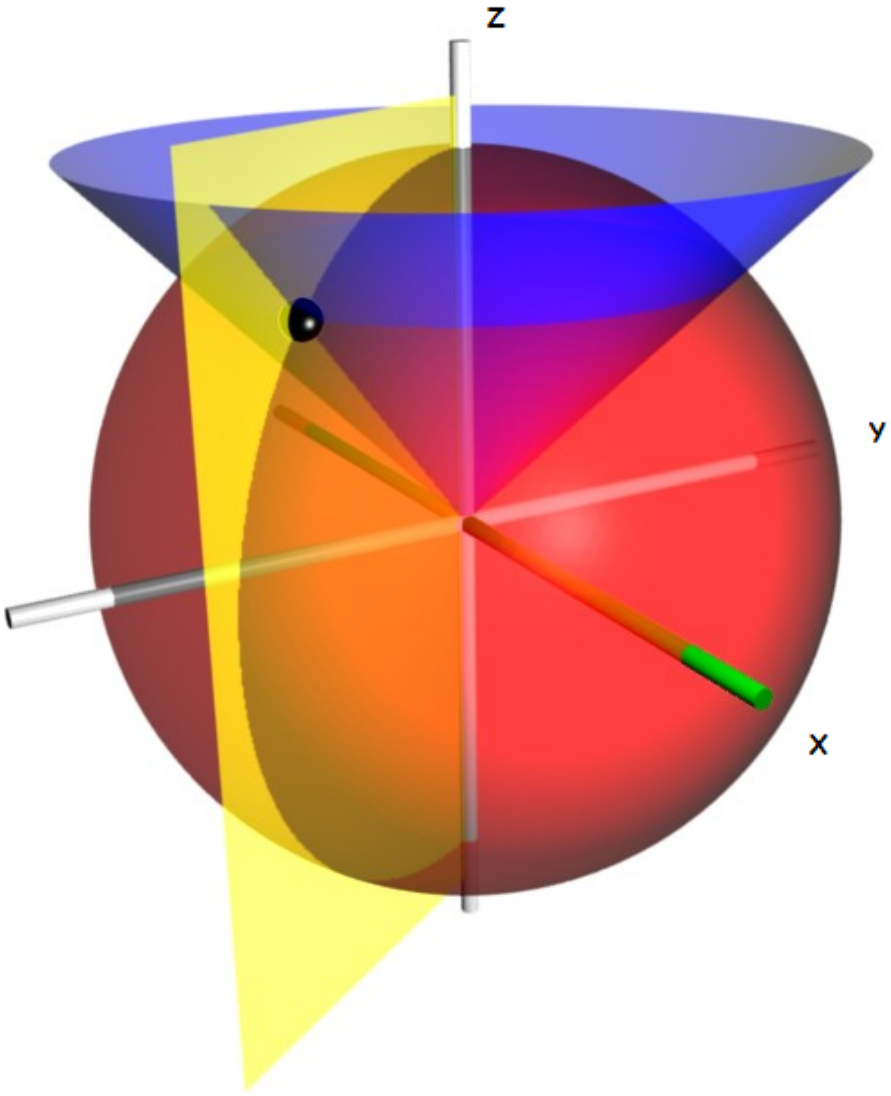


The skeleton is draw correctly and walks normally, or just as in the reference implementation.

3. Code must allow interactive specification of camera viewpoint:

Camera coordinates:

The camera coordinates are calculated using spherical coordinates (see above for more information on spherical coordinates).



The camera is represented by the black dot in this diagram. The blue cone specifies the **thetaCamera** angle. The yellow cone specifies the **phiCamera** angle. The red sphere specifies the **rCamera** distance. The camera moves around the origin on the red sphere. Once we have calculated the camera's coordinates we place it using the **gluLookAt** function. This function takes three coordinates as parameters: **eye** (position of the camera), **centre** (where the camera is looking at), and **up** (defines the vertical). The **up** is the Z-axis and will be defined as (0,0,1). The **centre** is (0,0,0) because the camera is looking at the origin and the **eye** will be the **xCamera,yCamera,zCamera** where:

```
xCamera = rCamera*sin(thetaCamera)*cos(phiCamera);
yCamera = rCamera*sin(thetaCamera)*sin(phiCamera);
zCamera = rCamera*cos(thetaCamera);
```

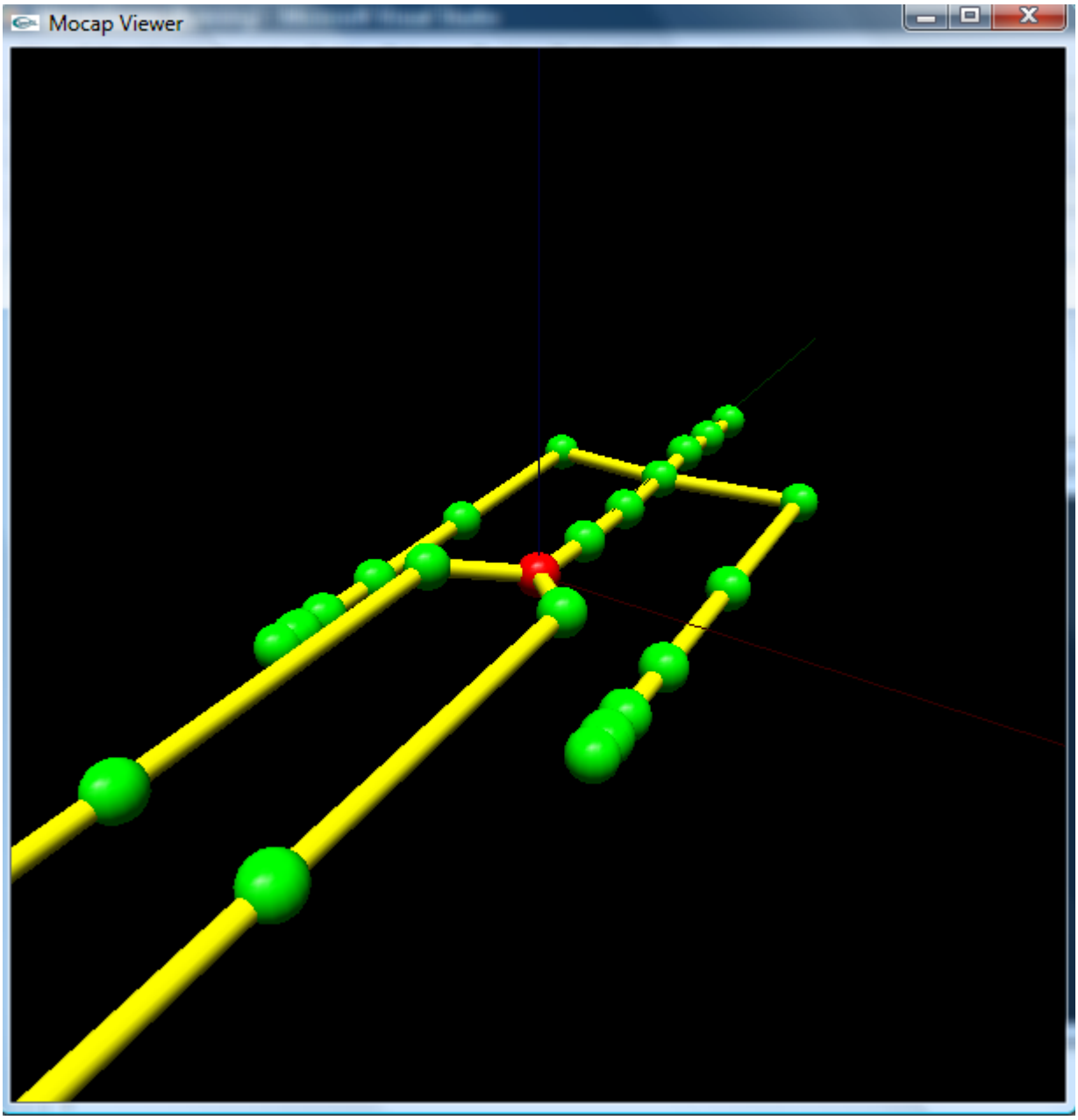
Theta and phi are in radians because the **sin()** and **cos()** functions from the math library are defined in radians.

Keyboard Interaction:

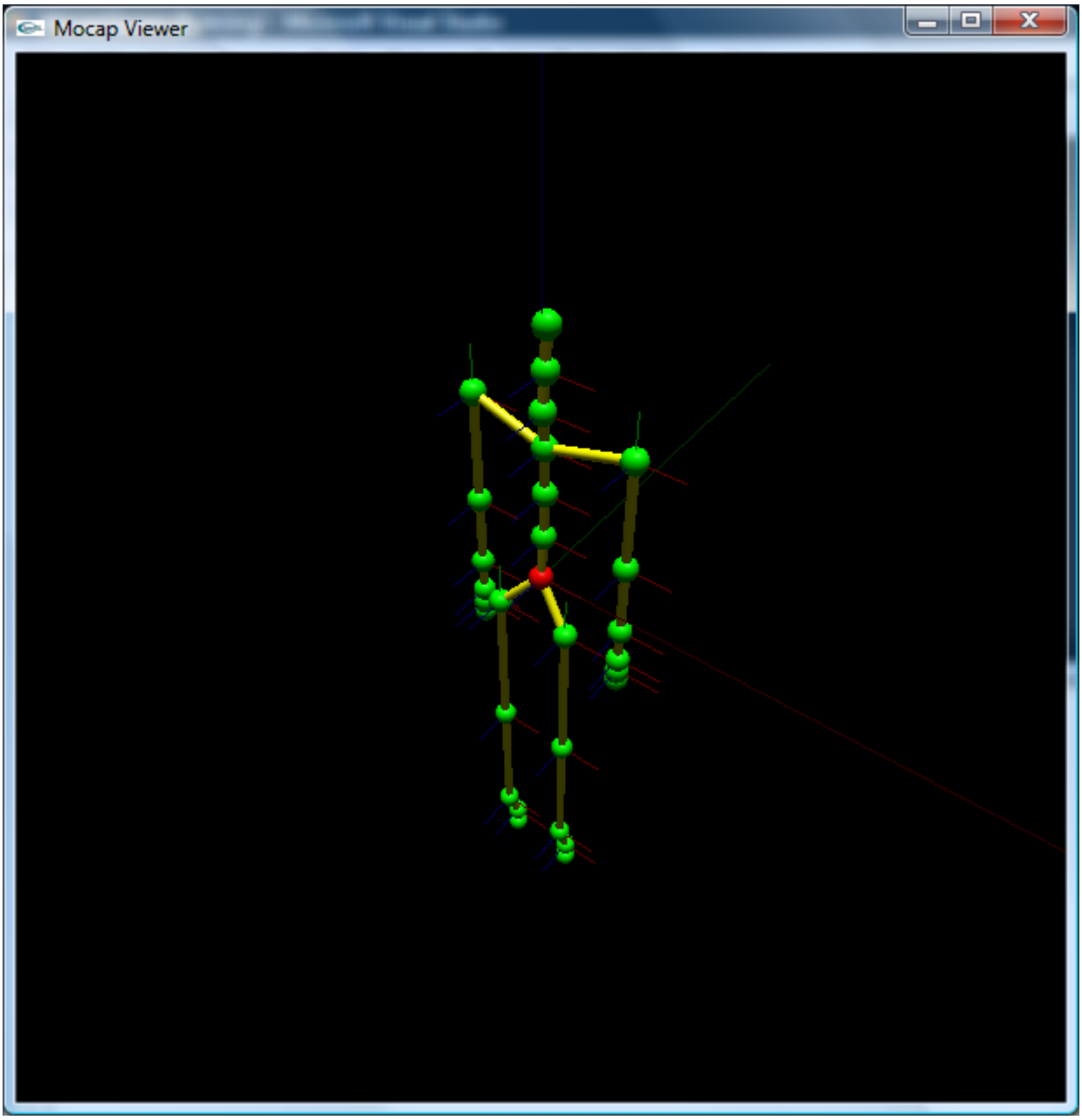
We will use the **keyboard()** callback from GLUT to handle the keyboard interaction. The camera polar coordinates can be modified by the user. I chose to map the keys 'wasd' to move the camera and 'qe' to zoom in and out. The 'wasd' keys will be familiar to a lot of users as they are the main keys used for moving camera or characters in games. 'w' and 's' enlarge or reduce the blue cone in the diagram (the theta angle), while 'a' and 'd' modify the phi angle (angle between the X-axis yellow plane in the diagram). I introduced a constant **CAMERA_SENS** which describes the camera sensitivity, i.e. the value by which the angles will be incremented or decremented.

Orientation of reference frame:

The implementation of the above code gives us this following scene:



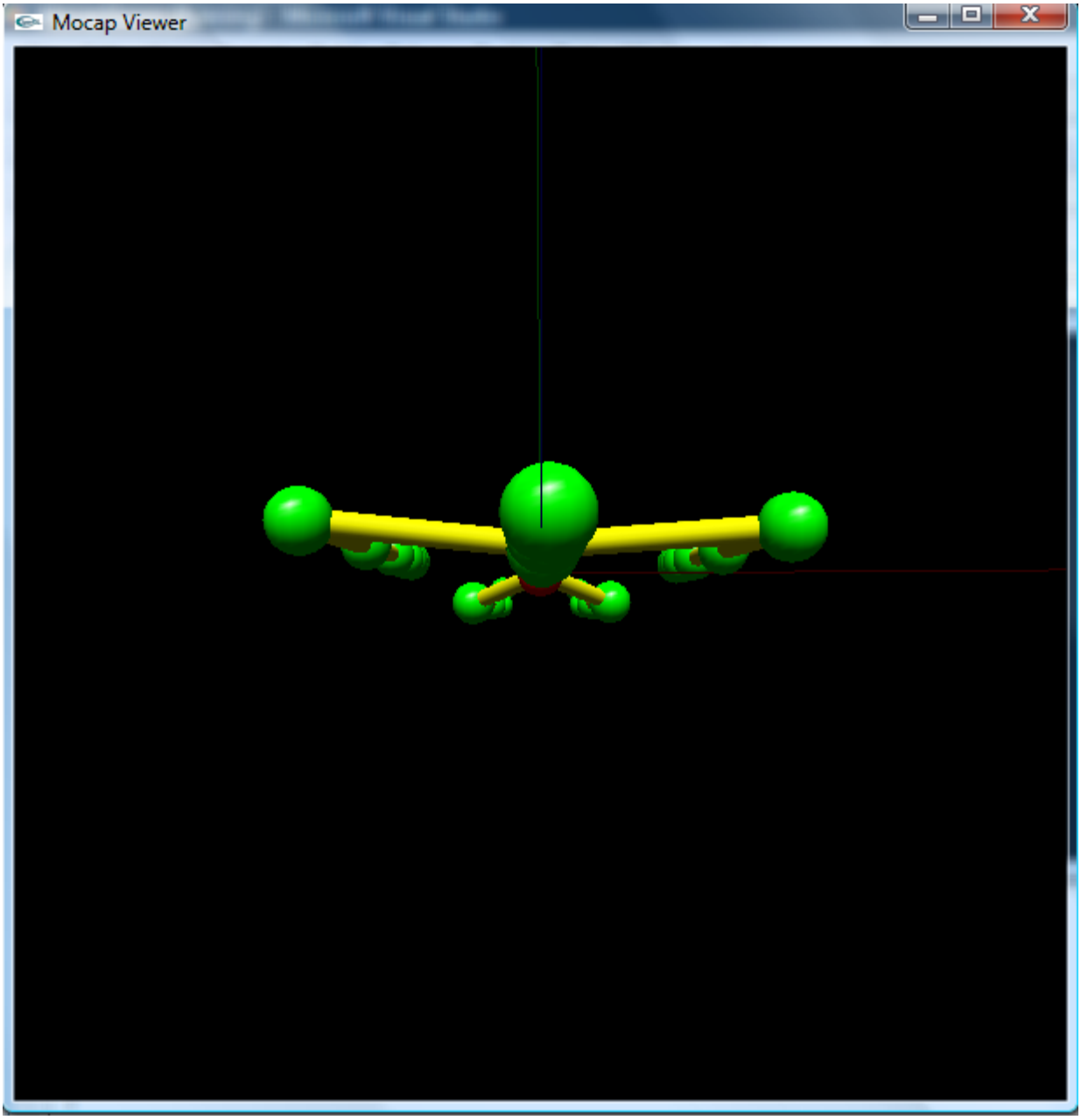
Here the skeleton is drawn horizontally instead of vertically. This is due to the skeleton being drawn up the Y-axis (Y-axis is the vertical) whereas in my calculation of the camera coordinates the Z-axis is the vertical. To fix this I rotate the reference frame by 90 degrees about the X-axis in my **drawSkeleton()** function, right before drawing the skeleton, so that the Y-axis "becomes" the Z-axis.



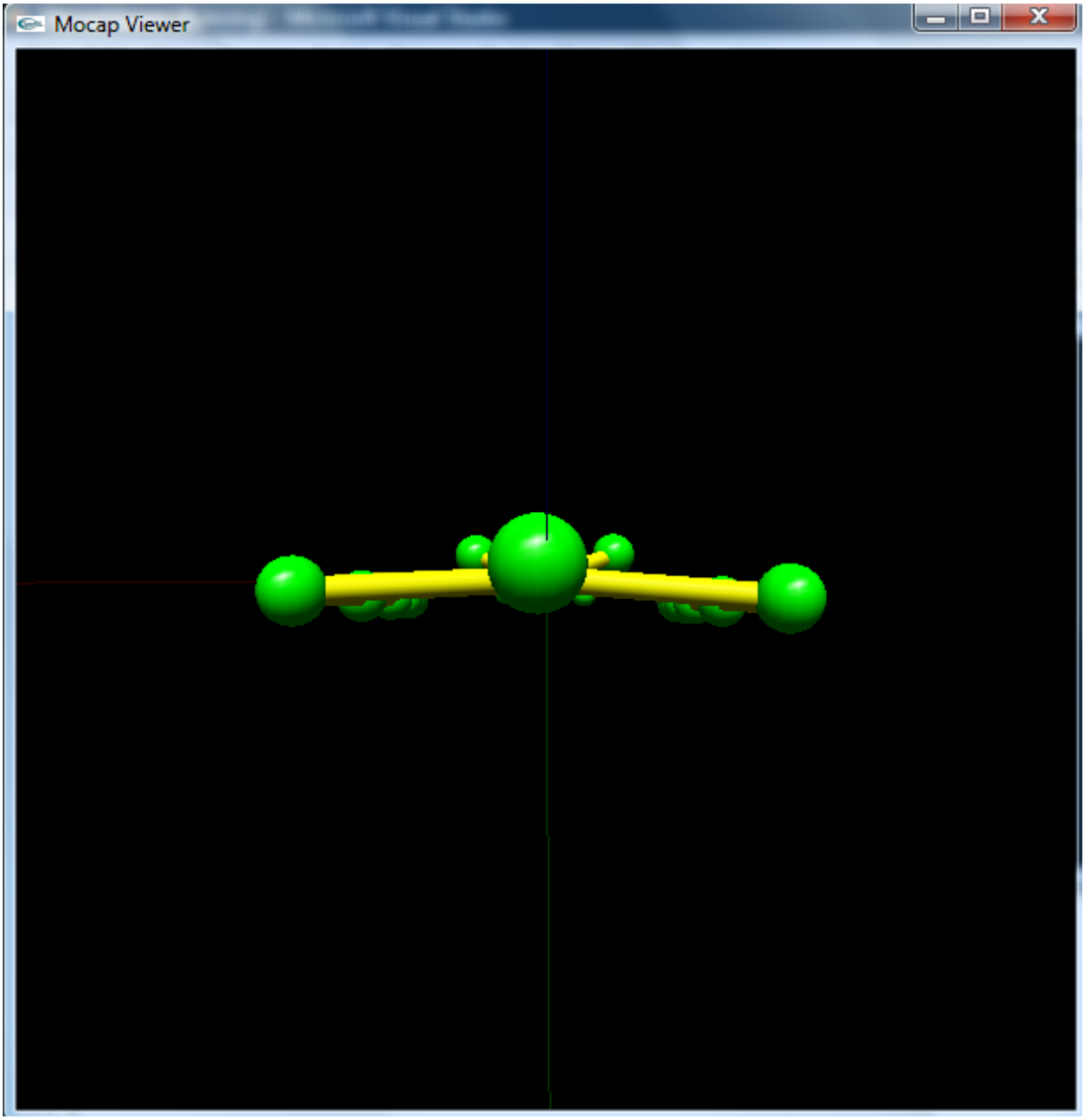
Here the skeleton is drawn correctly (up the Z-axis). The Y-axis is drawn in green and the Z-axis in blue. The only thing to note here is that Y becomes –Z because as you can see in the picture above the Y-axis draws away from us, whereas the Z-axis on the skeleton is coming towards us. The X-axis remains unchanged. So: X=X; Y=-Z; Z=Y;

Problems faced:

The main problem or bug that appears with this camera is the case when the user presses 'w' too many times so the **thetaCamera** becomes negative and the controls are inversed:



The camera's movements aren't limited. Here the camera should not go further up if the user presses on 'w'. It should not be able to show the skeleton from a bird's eye view or the theta angle will become negative and the controls switched (w to go down and s to go up). Here we are still rotating about theta so:
‘w’: theta -1;
‘s’: theta + 1;



Here the angles are reversed we are now rotating about -theta. Notice the x-axis is now on the left and the y-axis underneath. We have "gone over" the z-axis and are viewing the skeleton from the other side. The camera controls are now inversed.
‘w’: -theta-1 = theta +1
‘s’: = -theta + 1 = theta - 1
To fix this I simply put a limit on the theta angle by adding an ‘if’ statement after modifying the angle. If theta is negative (we have gone over Z) then add **CAMERA_SENS** . This just puts theta back to its previous value: when theta is negative or near 0 then pressing the ‘w’ key does nothing. I applied the same concept for the ‘s’ key, except this time if theta is greater than PI/2 (90 degrees) then set it back to PI/2. The camera is has now limited movement up and down: 0<theta<90, theta’s value can only be between 0 and 90 degrees.

4. Include simple visual aids – e.g. Texture mapped floor, or Reference Frames:

Loading Texture:

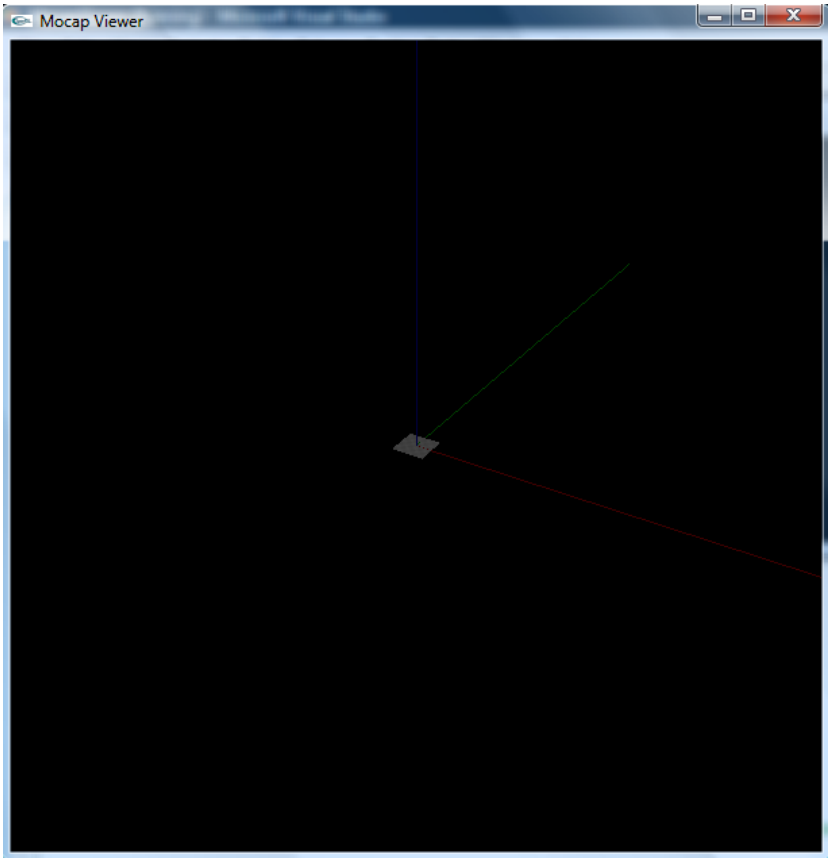
I first created a new function called **loadTexture()** which returns a texture (type **GLuint**). This gives a name to the texture and then sets its parameters such as the environment and if mip mapping is used.

Image texture:

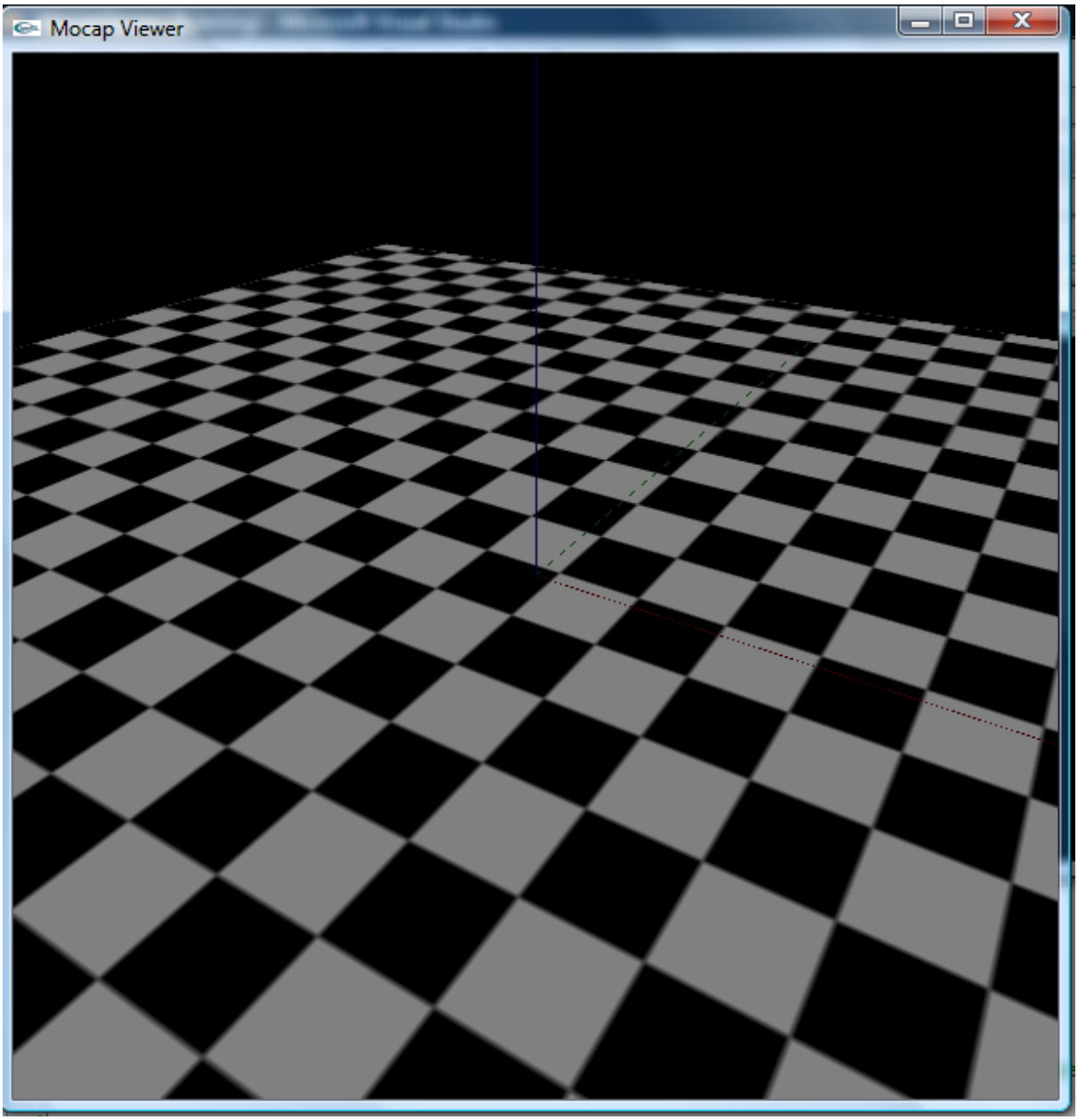
I tried to load a texture from a .jpg file and then set that as my floor, but I was unsuccessful in finding an adequate function and library that could do that.

Instead I used the code from the lecture notes and loaded a chequer board pattern as my floor by mapping the coordinates of the texture (**glTextCoord2f**) to the coordinates of a point in my 3D space. I do this using **GL_QUADS** because the floor is going to be a 2D square shape. The code looks like this, where 'w' is the width of the floor and 'h' the height:

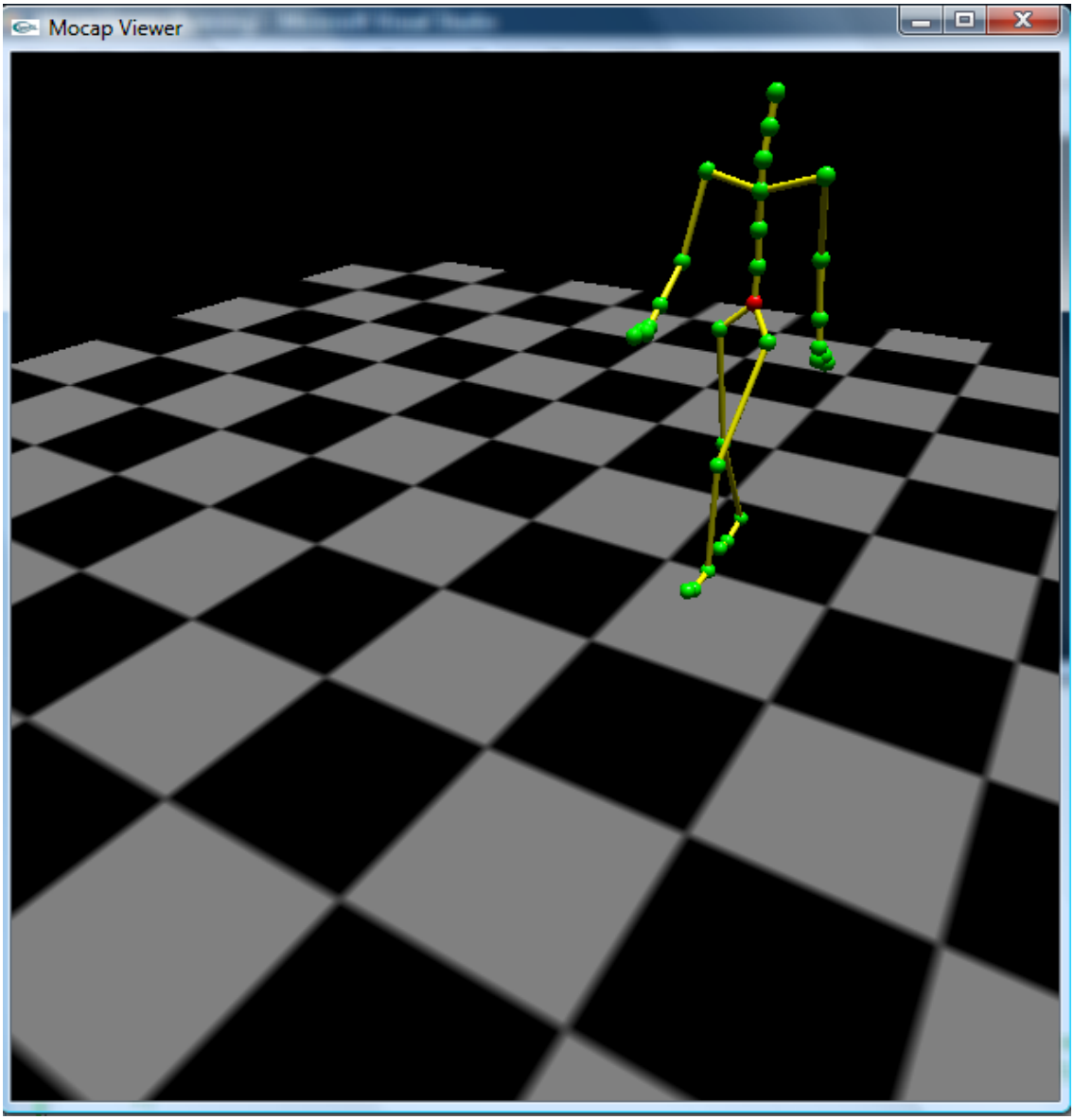
```
glBegin(GL_QUADS);
      glNormal3f(0,0,1);
      glTexCoord2f(0,0); glVertex3f(-w,-h,0);
      glTexCoord2f(1,0); glVertex3f(w,-h,0);
      glTexCoord2f(1,1); glVertex3f(w,h,0);
      glTexCoord2f(0,1); glVertex3f(-w,h,0);
glEnd();
```



Texture is drawn with width and height equal to 1. The texture display in this screenshot is much too small. The use of width and height for mapping the texture to a rectangle is actually going to stretch the texture to the specified size. It is not going to repeat the pattern. So if we specify a bigger width and height the texture will appear larger and the squares bigger:



The texture is stretched to fit a 100x100 square centred about the origin. The program is significantly slower and the squares are still too small. I therefore changed the texture size to 256 instead of 512 in the **loadTexture()** function. As a result the texture squares are bigger and our floor will be smaller which means the program will be a little faster:



Draw reference frames:

Draw the reference frames was relatively easy. I already had a function in my code called **drawReferenceFrame()** which draws the reference frame for the current identity matrix. The function takes a parameter that sets the length of the reference frame drawn. To draw the reference frames for each joint we simply have to call the **drawReferenceFrame()** function with a small length before doing any matrix transformations in the **drawJoints()** so that they are drawn for each joint.

5. Camera must track movement of skeleton:

The camera is currently centred on the origin using **gluLookAt** with parameter **centre** set at (0,0,0). All we have to do is set the **center** to the position of the root in the current frame and the camera will always be looking at the root of the skeleton and therefore following the skeleton.

Calculate root position:

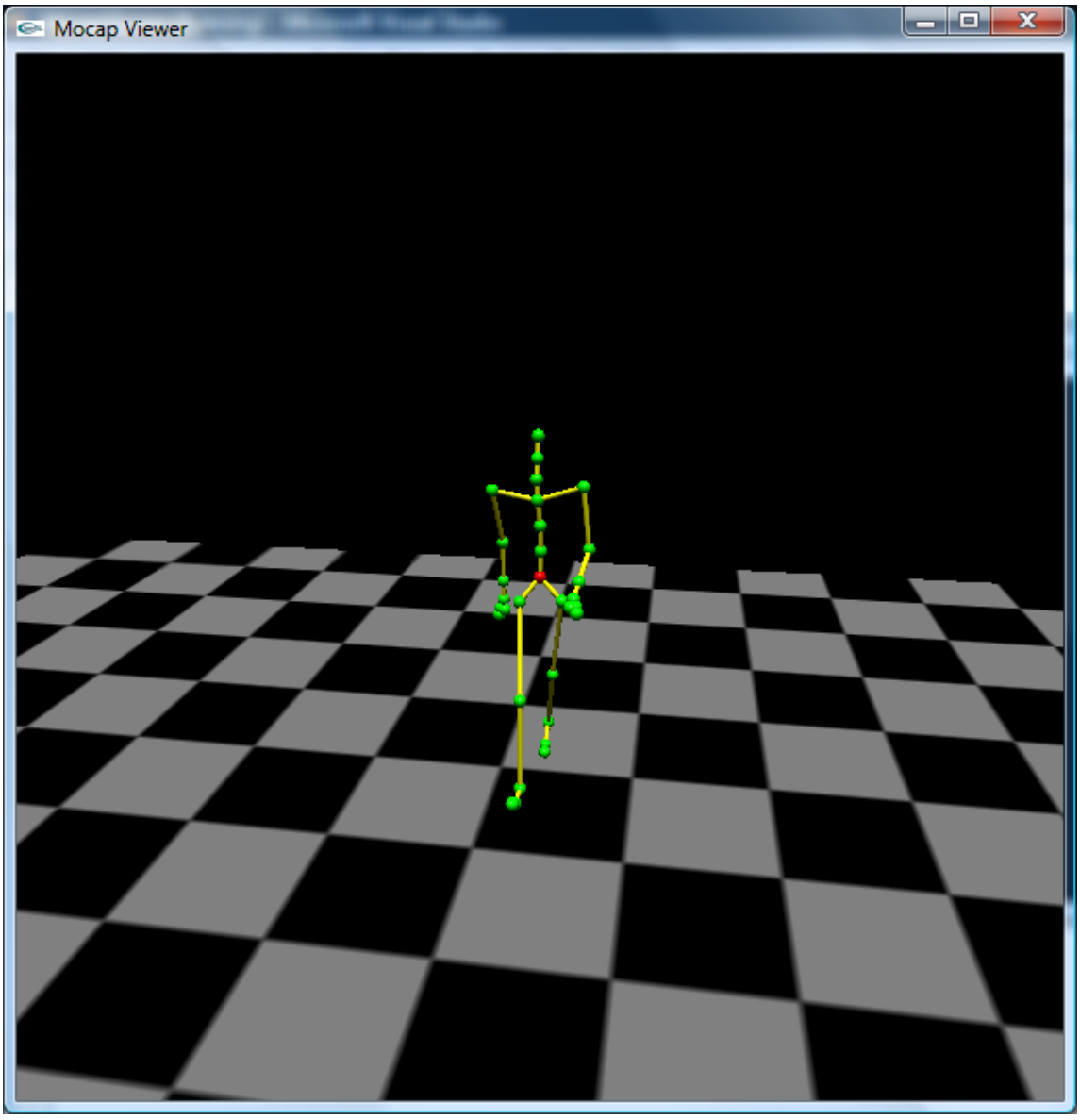
We introduce three new variables: **xRoot, yRoot, zRoot** which define the current position of the skeleton's root. The setting of those variable should look like this:

```
xRoot = gMo->root_pos[currentFrame].x;
yRoot = gMo->root_pos[currentFrame].y;
zRoot = gMo->root_pos[currentFrame].z;
```

As we have seen in the camera viewpoint implementation the Y-axis and the Z-axis are inversed where: X=X, Y=-Z and Z=Y, because the skeleton is originally drawn up the Y-axis and the **gMo->root_pos** is the original root positioning. We therefore have to inverse the Y and Z axis when we calculate the root position:

```
xRoot = gMo->root_pos[currentFrame].x;
yRoot = -gMo->root_pos[currentFrame].z;
zRoot = gMo->root_pos[currentFrame].y;
```

The camera should now be following the skeleton:



The only problem here is that the movement of the camera is still centred about the origin. We now have to recalculate its position so the movement is made around the skeleton.

Calculate camera position:

This is very simple. All we have to do is add the root position to the camera position, so that the camera will be moving around the skeleton instead of the origin.

```
xCamera+=xRoot; yCamera+=yRoot, zCamera+=zRoot
```

Now the user is able to rotate the camera around the skeleton.