(a) 0°, −90°, 0° (one possible answer)

We'll see more of Euler angles when we discuss animating the orientation of rigid objects in Chapter 6.

4.6.4 Viewing Transformation and Collision Detection in Animation

Our next program makes use of viewing transformations to simulate a moving camera in an animated environment. It also has another aspect of interest, particularly to those programming interactive applications such as games, namely, collision detection.

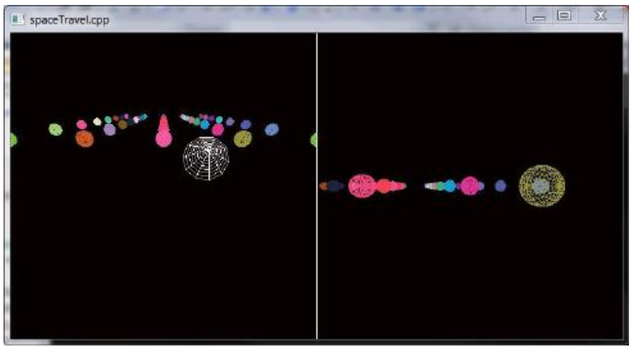


Figure 4.51: Screenshot of `spaceTravel.cpp`.

Experiment 4.31. Run `spaceTravel.cpp`. The left viewport shows a global view from a fixed camera of a conical spacecraft and 40 stationary spherical asteroids arranged in a 5 × 8 grid. The right viewport shows the view from a front-facing camera attached to the tip of the craft. See Figure 4.51 for a screenshot of the program.

Press the up and down arrow keys to move the craft forward and backward and the left and right arrow keys to turn it. Approximate collision detection is implemented to prevent the craft from crashing into an asteroid.

The asteroid grid can be changed in size by redefining `ROWS` and `COLUMNS`. The probability that a particular row-column slot is filled is specified as a percentage by `FILL_PROBABILITY` – a value less than 100 leads to a non-uniform distribution of asteroids. **End**

We'll discuss next the two most interesting aspects of `spaceTravel.cpp`: (a) the viewing transformation that defines the scene in the right viewport and (b) collision detection.

Viewing Transformation

The shape of the craft is defined by the `glutWireCone(5.0, 10.0, 10, 10)` statement; precisely, it is a cone of base radius 5 and height 10. The configuration of the spacecraft is specified by the values of $xVal$, $zVal$ and $angle$, all three global variables of `spaceTravel.cpp`. Figure 4.52(a) is a generic configuration in section along the xz-plane. The coordinates of the center of the craft's base are $(xVal, 0, zVal)$, while the angle its axis makes with the negative z-direction is $angle$. The middle A of the craft's axis will be of use in collision detection.

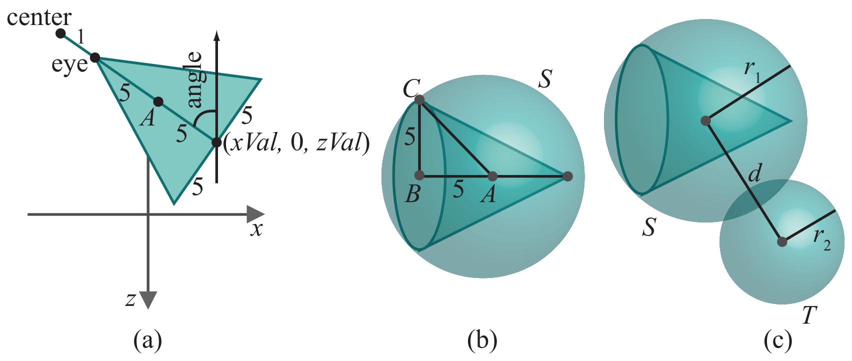


Figure 4.52: Spacecraft diagrams.

The camera for the right viewport is situated at the tip of the craft pointing straight ahead. It's straightforward trigonometry, now, to calculate the coordinates of eye, i.e., the tip of the craft, and of an imaginary point $center$ to which it points, located 1 unit ahead of the tip:

$$\begin{aligned} eye &= (\,xVal - 10\sin(angle),\ 0,\ zVal - 10\cos(angle)\,) \\ center &= (\,xVal - 11\sin(angle),\ 0,\ zVal - 11\cos(angle)\,) \end{aligned}$$

These equations for eye and $center$ explain the parameters of the `gluLookAt()` command for the right viewport.

Collision Detection

Collision detection as implemented in `spaceTravel.cpp` is simple though approximate. The spacecraft is enclosed in an imaginary bounding sphere S centered at the middle A of the cone's axis, with radius equal to the distance $|AC|$ from A to a point C on the boundary of its base. See Figure 4.52(b).

If B is the center of the base, then it follows from the dimensions of the cone that $|AB| = |BC| = 5$; therefore,

$$|AC| = \sqrt{|AB|^2 + |BC|^2} = \sqrt{50} = 7.071\ldots$$

Accordingly, we specify the radius of S to be 7.072 (slightly larger, in fact, than $|AC|$). The coordinates of the center A of S are obtained by trigonometry from Figure 4.52(a):

$$A = (\, xVal - 5\sin(angle),\ 0,\ zVal - 5\cos(angle)\,)$$

To detect collision between the spacecraft and an asteroid T, we detect instead collision between the craft's bounding sphere S and T. It's easy to determine if there is a collision between the two spheres S and T: compare the distance d between their centers with the sum $r_1 + r_2$ of their radii; there is collision if $d \le r_1 + r_2$ (e.g., as in Figure 4.52(c)), and not otherwise. This check is implemented in the routine `checkSphereCollision()`. This collision-detection test is approximate, in fact, conservative, as the craft's bounding sphere may intersect an asteroid even if the craft itself doesn't (as shown in Figure 4.52(c)).

The up and down arrow keys are programmed to move the craft a distance of 1 in either direction along its axis, and the left and right arrows to turn the craft an angle of 5°, *only if* there'll not be a collision with an asteroid in the new position (according to the conservative test above).

Exercise 4.55. (Programming) Modify `spaceTravel.cpp` as follows:

(a) Make the left viewport the view from the front of the spacecraft (currently, it is the right viewport).

(b) Make one of the asteroids the "big golden asteroid" by drawing it larger than the others and painting it suitably. Make it glow as well by oscillating the intensity of its color.

(c) Place a camera on the golden asteroid whose location is fixed but which rotates to track the spacecraft, i.e., its direction is pointed always toward the craft. Attach a tall antenna to the craft so that, even if it's obscured by other asteroids, at least the antenna will be visible from the big golden asteroid. Show the view from the golden asteroid's camera in the right viewport.

(d) When the spacecraft reaches the big golden asteroid, flash the text "You have found gold!".

Exercise 4.56. (Programming) Modify `spaceTravel.cpp` as follows:

(a) All the asteroids are currently colored spheres. Make them more interesting by using a few different FreeGLUT objects, e.g., cube, tetrahedron, octahedron, etc. You can also combine more than one object, e.g., one sphere on top of another, or design your own.

(b) Currently, the spacecraft moves interactively. Change this to program an automated tour which takes a fixed but zig-zag path through the asteroids and returns to the start position. Plan a path so that the craft comes close to a few interesting asteroids, visible in the right viewport. Pressing space should start/stop the movement.

(c) Currently, the camera on the craft always points straight ahead. Program occasional rotation of the camera, e.g., when the craft passes a strange asteroid, pan the camera to keep it in view.

Exercise 4.57. (Programming) Place a camera on top of the rolling ball of Exercise 4.27, pointing always down the plane. This camera does *not* rotate with the ball, but stays always at the top, so its motion is entirely linear. (How would you even install such a camera in real life? Well, that is a great thing about CG: you are entirely free from real-life constraints!)

Place a box just beyond the bottom of the plane so that the ball's camera sees an approaching object. Place an additional fixed camera on the box pointing at the plane to observe the ball. See Figure 4.53. Give a split-screen view as in `spaceTravel.cpp`.

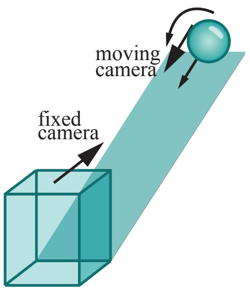


Figure 4.53: Ball rolling toward a box.

The following experiment is to whet your appetite for the topic of *frustum culling*, critical to the efficient rendering of complex scenes with large numbers of objects.

Experiment 4.32. Run `spaceTravel.cpp` with `ROWS` and `COLUMNS` both increased to 100. The spacecraft now begins to respond so slowly to key input that its movement seems clunky, unless, of course, you have a super-fast computer (in which case, increase the values of `ROWS` and `COLUMNS` even more). **End**

The reason for the degradation in the preceding experiment is that, every time an arrow key is pressed, OpenGL processes 10,000 asteroids, which is an enormous amount of computing. However, of these 10,000 only a few (about 100, or 1%) are ultimately rendered, as you can roughly count on the screen! The rest, of course, are outside the viewing frustum and clipped.

Unfortunately, by the time the decision to clip is made in the graphics pipeline, a large amount of computation has already been invested. Frustum culling is a technique to reduce this burden on OpenGL, whereby the programmer leverages her knowledge of the scene to pre-filter objects lying beyond the viewing frustum, not letting them into the pipeline at all.

We'll discuss frustum culling in detail in Section 6.1. There's really not much more by way of prerequisites needed to read that particular section though, so if you're anxious to learn this technique, which is so important in coding busy games and movies, feel free to jump right there.

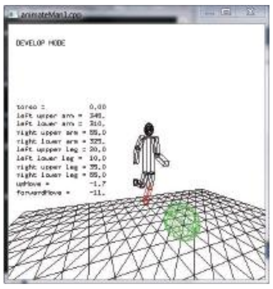
We are not done yet with animation, though, and have a bunch more fun code for you.

4.7 More Animation Code

4.7.1 Animating an Articulated Figure

Our next project is a "studio" to develop animation sequences for an articulated figure.

Experiment 4.33. Run `animateMan1.cpp`. This is a fairly complex program to develop a sequence of key frames for a man-like figure, which can subsequently be animated. In addition to its spherical head, the figure consists of nine box-like body parts which can rotate about their joints. See Figure 4.54. All parts are wireframe. We'll explain the program next. **End**



```
DEVELOP MODE

torso =              0.00
left upper arm =     340.
left lower arm =     310.
right upper arm =    55.0
right lower arm =    325.
left uppper leg =    20.0
left lower leg =     10.0
right upper leg =    35.0
right lower leg =    55.0
upMove =             -1.7
forwardMove =        -11.
```

Figure 4.54: Screenshot of `animateMan1.cpp`.

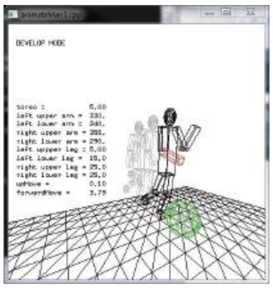
It's advisable to learn to use the program before studying the code. There are two modes, develop and animate, and the program starts in the develop mode with the man facing you with his currently highlighted part, the torso, colored red. The rest of the body is black. Press the space bar to cycle through the man's movable parts, successively highlighting each. There are nine movable parts, all OpenGL wire cubes: the torso, the upper and lower arms on either side, and the upper and lower legs on either side.

Rotate the currently highlighted part by pressing the page-up and page-down keys. To move the man as a whole press the left/right and up/down arrow keys. The angles at which the 9 movable parts are currently rotated, as well as the vertical and horizontal translational components of the man as a whole, are shown as text data in the window in develop mode.

While arranging the man into a desired configuration, you can rotate your own viewpoint by pressing 'r/R', or zoom in and out pressing 'z/Z'.

Once the first configuration is completed to your satisfaction, press 'n'. This creates a new configuration which cannot be seen immediately as it's a copy of the previous one. Press, say, the right arrow key to separate the new configuration from the previous one. The (current) new configuration is bright, while the other(s) are ghosted. Again, use the space key to select a part, the page-up and page-down keys to rotate that part, and the arrow keys to move the entire configuration until it is arranged suitably.

Press 'n' to create new configurations until the key frames sequence is complete. Figure 4.55 shows a screenshot part way through the develop mode. You can edit the sequence at any time as follows.



```
DEVELOP MODE

torso =              5.00
left upper arm =     330.
left lower arm =     340.
right upper arm =    355.
right lower arm =    290.
left uppper leg =    5.00
left lower leg =     15.0
right upper leg =    25.0
right lower leg =    25.0
upMove =             0.10
forwardMove =        3.75
```

Figure 4.55: Screenshot of `animateMan1.cpp` in develop mode.

Press the tab key to cycle through the sequence of configurations – the currently selected configuration is bright, while the rest ghosted. Press backspace to reset the currently selected configuration, delete to remove it altogether, or you can rearrange it using keys as already described.

When the key frames sequence is complete, pressing 'a' begins an animation which cycles through the programmer-created configurations. Pressing the up or down arrow keys speeds up or slows down the animation. Pressing 'a' again returns the program to develop mode.

Switching to animation mode also causes the program to write out to the file `animateManDataOut.txt` successive configurations of the animation sequence, stored currently in the vector `manVector`. Configuration are stored in successive lines of `animateManDataOut.txt`, each consisting of 11 floating point values – `partAngles[0]-[8]`, `upMove` and `forwardMove` – the same as are displayed on the screen in develop mode.

Experiment 4.34. Run `animateMan2.cpp`. This is simply a pared-down version of `animateMan1.cpp`, whose purpose is to animate the sequence of configurations listed in the file `animateManDataIn.txt`, likely generated from the develop mode of `animateMan1.cpp`. Press 'a' to toggle between animation on/off. As in `animateMan1.cpp`, pressing the up or down arrow key speeds up or slows down the animation. The camera functionalities via the keys 'r/R' and 'z/Z' remain as well. Think of `animateMan1.cpp` as the studio and `animateMan2.cpp` as the movie.

The current contents of `animateManDataIn.txt` cause the man to do a handspring over the ball. Figure 4.56 is a screenshot. **End**

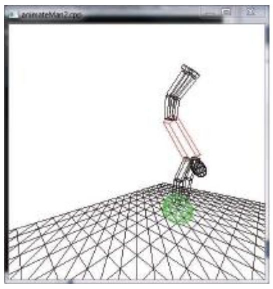


Figure 4.56: Screenshot of `animateMan2.cpp`.

Now let's look at the code of `animateMan1.cpp`. From an OpenGL point of view, most interesting possibly is the drawing of a configuration by the function `Man::draw()`. The best way to understand it is to analyze the successive placement of parts. We'll do this our usual way of deconstructing a program by first commenting out most of it and then restoring code piece by piece.

Accordingly, first comment out all parts except the torso as below:

```
// Function to draw man.
void Man::draw()
{
   if (highlight||animateMode) glColor3fv(highlightColor);
   else glColor3fv(lowlightColor);

   glPushMatrix();

   // Up and forward translations.
   glTranslatef(0.0, upMove, forwardMove);

   // Torso begin.
   if (highlight && !animateMode) if (selectedPart == 0)
      glColor3fv(partSelectColor);

   glRotatef(partAngles[0], 1.0, 0.0, 0.0);

   glPushMatrix();
   glScalef(4.0, 16.0, 4.0);
   glutWireCube(1.0);
   glPopMatrix();
   if (highlight && !animateMode) glColor3fv(highlightColor);
   // Torso end.
   /*
   // Head begin.
   .
   .
   .
   // Right upper and lower leg with foot end.
   */

   glPopMatrix();
}
```

Next, uncomment the head:

```
// Function to draw man.
void Man::draw()
{
   if (highlight||animateMode) glColor3fv(highlightColor);
   else glColor3fv(lowlightColor);

   glPushMatrix();

   // Up and forward translations.
   glTranslatef(0.0, upMove, forwardMove);

   // Torso begin.
   if (highlight && !animateMode) if (selectedPart == 0)
      glColor3fv(partSelectColor);

   glRotatef(partAngles[0], 1.0, 0.0, 0.0);

   glPushMatrix();
   glScalef(4.0, 16.0, 4.0);
   glutWireCube(1.0);
   glPopMatrix();
   if (highlight && !animateMode) glColor3fv(highlightColor);
   // Torso end.

   // Head begin.
   glPushMatrix();

   glTranslatef(0.0, 11.5, 0.0);
   glPushMatrix();
   glScalef(2.0, 3.0, 2.0);
   glutWireSphere(1.0, 10, 8);
   glPopMatrix();

   glPopMatrix();
   // Head end.

   /*
   // Left upper and lower arm begin.
   .
   .
   .
   // Right upper and lower leg with foot end.
   */

   glPopMatrix();
}
```

Continue – as you successively uncomment each body part, it'll be clear how it's being placed with respect to existing ones.

The creation of the `camera` as an object of the `Camera` class may be of interest as well and we'll leave the reader to relate the parameter values of the `gluLookAt()` command to the member variables `viewDirection` and `zoomDistance` of the `Camera` class.

Much of the rest of the code consists simply of managing and using `manVector`, which stores the sequence of configurations.

Remark 4.18. Even though he himself is 3D, the man moves and rotates his parts always parallel to the yz-plane, so he's not really capable of 3D motion!

Exercise 4.58. (Programming) Use `animateMan*.cpp` to animate a character kicking a football.

Exercise 4.59. (Programming) Enhance `animateMan*.cpp`:

(a) The character's body parts, except for the head, are currently all cubes. Make them more realistically rounded using cylinders.

(b) Add movement to the character's feet, which are currently fixed with respect to his lower legs. Give him movable hands as well.

(c) As remarked earlier, all the character's movements are currently parallel to a single plane. Enhance to true 3D.

Exercise 4.60. (Programming) Stick a camera to the front of the man's head and give a split-screen view of what he sees as he advances through an animation sequence and what is seen from a separate fixed camera focused on him.

Exercise 4.61. (Programming) By scaling individual body parts, create a second character who looks different from the first, though with identical functionality. Make a simple movie with the two.

It would be particularly effective in such a sequence to occasionally switch to a camera located in front of either one of their heads, to record how one sees the other.