BLG444E

Computer Graphics

2014 Spring

Report of Project 4

Date of Submission : 05.05.2014

040100118 Volkan İlbeyli
040100043 Mert Tepe

Instructor : Uluğ Bayazıt

The two of us did the entire project on our own and then in the end, combined our work into one project for the purpose of either of us having learnt the basics of OpenGL and 3D graphics.

Task 1: Drawing the Robots Using A Scene Graph

Code migration is completed as instructed in the asst4_snippets.cpp file. To build the robot, the following adjustments are made in `constructRobot()` function as to draw the legs and the head:

```
  const double ARM_LEN = 0.7,
               ARM_THICK = 0.25,
               TORSO_LEN = 1.5,
               TORSO_THICK = 0.25,
               TORSO_WIDTH = 1,
               LEG_LEN = 0.9,
               LEG_WIDTH = 0.3,
               LEG_THICK = 0.3,
               HEAD_WIDTH = 0.6,
               HEAD_HEIGHT = 0.5,
               HEAD_THICK = 0.28;
  const int NUM_JOINTS = 10,
            NUM_SHAPES = 10;

  JointDesc jointDesc[NUM_JOINTS] = {
    {-1}, // torso
    {0,  TORSO_WIDTH/2, TORSO_LEN/2, 0},  // upper right arm
    {1,  ARM_LEN, 0, 0                 },  // lower right arm
    {0,  -TORSO_WIDTH/2, TORSO_LEN/2, 0}, // upper left arm
    {3,  -ARM_LEN, 0, 0},                  // lower left arm
    {0,  TORSO_WIDTH/2, -TORSO_LEN/2, 0},  // upper right leg
    {5,  0, -LEG_LEN, 0                 },   // lower right leg
    {0,  -TORSO_WIDTH/2, -TORSO_LEN/2, 0}, // upper left leg
    {7,  0, -LEG_LEN, 0                 },  // lower left leg
    {0,  0, TORSO_LEN/4, 0}             // head
  };

  ShapeDesc shapeDesc[NUM_SHAPES] = {
    {0, 0,         0, 0, TORSO_WIDTH, TORSO_LEN, TORSO_THICK, g_cube}, // torso
    {1, ARM_LEN/2, 0, 0, ARM_LEN, ARM_THICK, ARM_THICK, g_cube}, // upper right arm
    {2, ARM_LEN/2, 0, 0, ARM_LEN, ARM_THICK, ARM_THICK, g_cube}, // lower right arm
    {3, -ARM_LEN/2, 0, 0, ARM_LEN, ARM_THICK, ARM_THICK, g_cube}, // upper left arm
    {4, -ARM_LEN/2, 0, 0, ARM_LEN, ARM_THICK, ARM_THICK, g_cube}, // lower left arm
    {5, LEG_WIDTH/2, 0, 0, LEG_WIDTH, LEG_LEN, LEG_THICK, g_cube}, // upper right leg
    {6, LEG_WIDTH/2, 0, 0, LEG_WIDTH, LEG_LEN, LEG_THICK, g_cube}, // lower right leg
    {7, -LEG_WIDTH/2, 0, 0, LEG_WIDTH, LEG_LEN, LEG_THICK, g_cube}, // upper left leg
    {8, -LEG_WIDTH/2, 0, 0, LEG_WIDTH, LEG_LEN, LEG_THICK, g_cube}, // lower left leg
    {9, 0, TORSO_LEN/2, 0, HEAD_WIDTH, HEAD_HEIGHT, HEAD_THICK, g_cube}
  };
```

Since the drawing technique and the data structures changed, we needed to implement translation/rotation manipulations function `doMtoOwrtA()` accordingly to take `SgRbtNode` as parameters as follows:

```
static void doMtoOwrtA(RigTForm m, shared_ptr<SgRbtNode>& o, shared_ptr<SgRbtNode>& a){
  RigTForm O = o->getRbt();
  RigTForm AA = a->getRbt();
  RigTForm A = transFact(O)*linFact(AA);
  o->setRbt(A * m * inv(A) * O);
}
```

Task 2: Keyframes

Animation variables are introduced:

```
//////////////////// Animation variables
vector<shared_ptr<SgRbtNode> > rbtNodes;
list<vector<RigTForm> > keyframes;
list<vector<RigTForm> >::iterator currentFrame = keyframes.end(); //undefined initially

static int g_msBetweenKeyFrames = 2000;
static int g_animateFramesPerSecond = 60;

bool isPlaying = false;
```

First vector variable is a temporary container to manipulate / store the current scene graph's rigid body transforms and is populated using the `dumpSgRbtNodes(g_world, rbtNodes);` function when necessary.

The keyboard interactions are implemented as they are described. Since they are fairly easy to implement and understand, no details will be given here for the easier ones. The most used key (n) will be explained, however.

```
case 'n':
      {
        cout << "New keyframe created from the current state" << endl;
        // clear Nodes and get the current state of the world
        rbtNodes.clear();
        dumpSgRbtNodes(g_world, rbtNodes);

        // container to push back to keyframe list
        vector<RigTForm> frame;

        // populate tehe container
        for(unsigned i=0; i<rbtNodes.size() ; i++)
          frame.push_back(rbtNodes[i]->getRbt());


        if(currentFrame != keyframes.end()){
          list<vector<RigTForm> >::iterator whereTo = currentFrame;
          whereTo++;
          keyframes.insert(whereTo, frame);
          currentFrame++;
        }
        else{ //keyframe isn't defined
          // save current state into keyframes
          keyframes.push_back(frame);

          // assign current frame into the newly pushed frame
          currentFrame = keyframes.begin();
        }

      }
      break;
```

First, the vector holding the (then-current) scene graph is cleared, so that the current scene graph can be pushed into the vector using `dumpSgRbtNodes(g_world, rbtNodes);` function call. Then, the values of the pointers are pushed into a new vector representing the current frame. If the current frame variable is defined, the captured scene frame is inserted after the place where current frame variable points to, and then the current frame variable is incremented to point to the newly inserted frame. If the current frame variable is not defined, then the newly captured frame is pushed back into the keyframes list.

When reading in a file for an animation, a format has to be maintained. For this program, the input file should contain the following lines:

```
6          // keyframe count
22         // RigTForm count per keyframe
x y z      // translation
w x y z    // rotation (quaternion)
x y z
w x y z
.
.
.
```

When 'I' is pressed, the user is prompted the name of the input file. When writing out, the program will output the file as 'out.txt'.

Task 3 & 4: Linear Interpolation and The Animation

When 'y' is pressed, the animation will start playing. While the animation is playing, pressing 'y' again will not trigger anything. To start the animation, `animateTimerCallback(0)` is called as described in the book.

```
static void animateTimerCallback(int ms){
  float t = (float)ms/(float)g_msBetweenKeyFrames;

  bool endReached = interpolateAndDisplay(t);
  if(!endReached){
     glutTimerFunc(1000/g_animateFramesPerSecond, animateTimerCallback,
                   ms + 1000/g_animateFramesPerSecond);
  }
  else{
    cout << "Animation ended" << endl;
    isPlaying = false;
  }
}
```

The animation will play until the end after the call, by checking the variable `endReached` which is assigned by the `interpolateAndDisplay(t)` function where the interpolation function `sLerp()` is called. This function will determine the keyframe interval based on the t value and interpolate transforms according to that keyframe interval [rbt0, rbt1].

```
bool interpolateAndDisplay(float t){
    float alpha = t - floor(t);

    if( floor(t)+1 < keyframes.size() ){

      // indexes of frames are passed to sLerp
      // which is lerp & slerp --> sLerp
      sLerp(floor(t), floor(t)+1, alpha);

      glutPostRedisplay();
      return false;
    }
    else
      return true;
}
```

In `sLerp()` function, current and next frame is determined first. Then, the pointers to rigid body transforms in the current scene graph are acquired using dump function provided by the assignment. After the pointers are acquired, each of them are updated using lerp and slerp.

```
void sLerp(int currI, int nextI, float alpha){

    // since pdf instructed us to keep the keyframes as
    // a list, we have to iterate every time to get the
    // array of keyframes in the corresponding indexes
    // instead of keeping them as arrays and reach them instantly
    vector<RigTForm> curr = getFromIndex(currI);
    vector<RigTForm> next = getFromIndex(nextI);

    rbtNodes.clear();
    dumpSgRbtNodes(g_world, rbtNodes);

    for(unsigned i=0; i<rbtNodes.size(); ++i){
      //RigTForm currentTForm = rbtNodes[i]->getRbt();
      RigTForm currentTForm = curr[i];

      // lerp translation
      const Cvec3 lerp = curr[i].getTranslation()*(1-alpha) + next[i].getTranslation()*alpha;
      currentTForm.setTranslation(lerp);

      //slerp rotations
      const Quat slerp = pow(cn( next[i].getRotation() * inv(curr[i].getRotation()) ), alpha)
                           *curr[i].getRotation();
      currentTForm.setRotation(slerp);

      rbtNodes[i]->setRbt(currentTForm);
    }

    return;
}
```

Lerp for translations is pretty straightforward blending between two frames. However slerp for quaternions are a little bit more sophisticated due to quaternions. The formula given in assignment is used to implement slerp

$$slerp(q_0, q_1, \alpha) := \left(cn(q_1 {q_0}^{-1})\right)^{\alpha} q_0$$

```
const Quat slerp = pow( cn( next[i].getRotation() * inv(curr[i].getRotation()) ), alpha)
                     *curr[i].getRotation();
```

Where cn() is:

```
Quat cn(Quat q){
  if(q[0] < 0)
    q *= -1;

  return q;
}
```

And the quaternion power operator implemented in quat.h is

```
inline Quat pow(const Quat& q, float alpha){
  Quat qu;

  qu[0] = cos( acos(q[0])*alpha );
  qu[1] = sin( asin(q[1])*alpha );
  qu[2] = sin( asin(q[2])*alpha );
  qu[3] = sin( asin(q[3])*alpha );

  return qu;
}
```

We couldn’t manage to use atan2() function, but instead, using the definition of power operator (the angle being multiplied by the power) we defined the function as above. It simply takes the angle theta by acos() and asin() functions, and then multiply it by alpha (the power), and finally assigning the quaternion elements accordingly.