BLG444E

Computer Graphics

2014 Spring

Report of Project 3

Date of Submission : 11.04.2014

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.

Adjustment to Assignment 2

We realized we missed some manipulation state as well as we could better-structure the code from previous assignment. Before getting into quaternions, here's what we've fixed:

We used to invert the translations when a mouse button was clicked. We changed those lines back to normal code and inverted the translations using `inv()` function as follows:

Here we fixed mouse click parts back to normal

```
Matrix4 m;
if (g_mouseLClickButton && !g_mouseRClickButton) { // left button down?
   m = Matrix4::makeXRotation(-dy) * Matrix4::makeYRotation(dx);
}
.
.
```

And then we inverted manipulations using inversion function. Then completed missing relative manipulation states which can be seen in the nested switch statements.

```
switch(manipState){ // manipulation state: 0=sky, 1=redCube, 2=greenCube
case 0:     // sky camera is manipulated
   if(viewState == 0){    // sky camera is the eye
      if(worldSkyFrame){  // if world-sky frame

          Matrix4 A = transFact(m)*linFact(inv(g_skyRbt));

          //doMtoOwrtA(inv(m), g_skyRbt, A);
          // here we cannot doMtoOwrtA since the function takes transfact(#2 param)
          // which corresponds to transfact(g_skyRbt) while we prefer transfact(m)
          // therefore we need to define our frame and do the multiplication on our own
          // in order to achieve circulating motion around the world frame.

          // translation is inverted when fed to the function
            g_skyRbt = A*inv(m)*inv(A)*g_skyRbt;
         }
         else{    // if sky-sky frame
            // translation is inverted before feeding into function
            // therefore it will be inverted back to normal when fed to function
            // thus only the rotations will be inverted
            m = transFact(inv(m)) * linFact(m);
            doMtoOwrtA(inv(m), g_skyRbt, g_skyRbt);
        }
    }
   break;
case 1:     // red cube is manipulated
   switch(viewState){
   case 0:   // eye = sky-camera
      doMtoOwrtA(m, g_objectRbt[0], g_skyRbt);// cube-sky frame
      break;
    case 1:   // eye = red-cube itself
       m = transFact(inv(m)) * linFact(m);  // keep translation un-inverted
       doMtoOwrtA(inv(m), g_objectRbt[0], g_objectRbt[0]);
       break;
    case 2:   // eye = green-cube
      doMtoOwrtA(m, g_objectRbt[0], g_objectRbt[1]); //cube i - cube j frame
      break;
    }
```

```
      break;
    case 2:     // green cube is manipulated
      switch(viewState){
     case 0:   // eye = sky-camera
       doMtoOwrtA(m, g_objectRbt[1], g_skyRbt);// cube-sky frame
       break;
     case 1:   // eye = red-cube
       doMtoOwrtA(m, g_objectRbt[1], g_objectRbt[0]);
       break;
     case 2:   // eye = green-cube itself
       m = transFact(inv(m)) * linFact(m);  // keep translation un-inverted
       doMtoOwrtA(inv(m), g_objectRbt[1], g_objectRbt[1]); //cube i - cube j frame
       break;
     }
    break;
  }
```

Doing all this was important because for quaternion implementation, we have to overload `doMtoOwrtA()` function. Now that we had fixed what was missing / messy from assignment 2, we are good to go for rigid body implementation.

Rigid Body Implementation

We first implemented constructors as unless a parameter is specified, identity value is assigned accordingly to class fields.

rigtform.h

```
  RigTForm(const Cvec3& t, const Quat& r) : t_(t), r_(r){
    //TODO (done)
  }

  explicit RigTForm(const Cvec3& t) : t_(t) {
    // TODO (done)
    r_ = Quat(); // identity quaternion
  }

  explicit RigTForm(const Quat& r) : r_(r) {
    // TODO (done)
    t_ = Cvec3();   // identity vector3
  }
```

Next, we implemented multiplication and inversion functions as described in the book.

```
  Cvec4 operator * (const Cvec4& a) const {
    // TODO (done)
   return r_*a + Cvec4(t_, 0);
  }

  RigTForm operator * (const RigTForm& a) const {
    // TODO (done)

   // convert Cvec3's of RigTForms into Cvec4s
   Cvec4 t4(t_[0], t_[1], t_[2], 1);
   Cvec4 at4(a.t_[0], a.t_[1], a.t_[2], 1);

   // find translational result
   Cvec4 tResult = t4 + r_* at4;
   Cvec3 tResult3(tResult[0], tResult[1], tResult[2]);

   // return T = t1 + r1*t2 || R = r1*r2
   return RigTForm(tResult3, r_ * a.r_);
  }
```

```
};

inline RigTForm inv(const RigTForm& tform) {
   // TODO (done)

  // make transform component Cvec4
  Cvec4 translation(tform.getTranslation(), 1);
  Cvec4 Tresult = inv(tform.getRotation()) * -translation;

  // convert result back to Cvec3
  Cvec3 Tresult3(Tresult[0], Tresult[1], Tresult[2]);

  // return T = -inv(r)*t || R = inv(r)
  return RigTForm(Tresult3, inv(tform.getRotation()));

}
```

Finally, the rigid body transformation to matrix4 conversion is implemented.

```
inline Matrix4 rigTFormToMatrix(const RigTForm& tform) {
   // TODO (done)

  // fix rotational part of matrix4
  Matrix4 m = quatToMatrix(tform.getRotation());

  // fix translational part of matrix4
  m(0,3) = tform.getTranslation()[0];
  m(1,3) = tform.getTranslation()[1];
  m(2,3) = tform.getTranslation()[2];

   return m;
}
```

Now that we have our fundamental functions implemented, it's time for replacing matrix4's in main code with RigTForm instances. In the end, we convert the RigTForm into matrix4 when feeding MVM into shader.

ass3.cpp

Object rigid body transformations are defined.

```
static RigTForm g_skyRbt = RigTForm( Cvec3(0.0, 0.25, 4.0) );
//static Matrix4 g_skyRbt = Matrix4::makeTranslation(Cvec3(0.0, 0.25, 4.0));

static RigTForm g_objRbt[2] = { RigTForm( Cvec3(-.85,0,0) ), RigTForm( Cvec3(.85, 0, 0) ) };
//static Matrix4 g_objectRbt[2] = {Matrix4::makeTranslation(Cvec3(-.85,0,0)),
Matrix4::makeTranslation(Cvec3(.85,0,0))};  // currently only 1 obj is defined
```

`drawStuff()` function updated to make use of `RigTForms` instead of `Matrix4s` and finally when assigning `MVM`, `RigTForms` are converted to `Matrix4s`.

```
  // assign eye rigid body matrix;
  //Matrix4 eyeRbt;
  RigTForm eyeRbt;
  switch(viewState){
  case 0:
     eyeRbt = g_skyRbt;
     break;
  case 1:
     eyeRbt = g_objRbt[0];
    break;
  case 2:
     eyeRbt = g_objRbt[1];
     break;
  }
```

```
const RigTForm invEyeRbt = inv(eyeRbt);

 .
 .
 .

// draw ground
// ===========
//
//const Matrix4 groundRbt = Matrix4();  // identity
const RigTForm groundRbt;
Matrix4 MVM = rigTFormToMatrix(invEyeRbt * groundRbt);
 .
 .
```

Matrix conversion is the same for drawing the cubes. Similarly for the rest of the document, all `Matrix4`'s associated with sky camera and cubes are converted to `RigTForm` and operated on accordingly.

Finally, the `doMtoOwrtA()` function is overridden as to work with `RigTForm` parameter. Again, all the multiplications are done with `RigTForms`, not `Matrix4s`.

Arcball Implementation

Global variables for arcball sphere are initialized.

```
// arcball
static float g_arcballScreenRadius = 0.25*min(g_windowWidth, g_windowHeight);
static float g_arcballScale;

// Vertex buffer and index buffer associated with the ground and cube geometry
static shared_ptr<Geometry> g_ground, g_cube, g_sphere;


static Cvec3f g_objectColors[3] = {Cvec3f(1, 0, 0), Cvec3f(.051,.75,.137), Cvec3f(0, 0, 1)};

static RigTForm g_sphereRbt = RigTForm();
```

A sphere geometry is defined similar to that of cubes.

```
static void initSphere(){
  int ibLen, vbLen;
  getSphereVbIbLen(20, 20, vbLen, ibLen);

  // Temporary storage for sphere geometry
  vector<VertexPN> vtx(vbLen);
  vector<unsigned short> idx(ibLen);

  makeSphere(1, 20, 20, &vtx[0], &idx[0]);
  g_sphere.reset(new Geometry(&vtx[0], &idx[0], vbLen, ibLen));
}
```

Draw function is updated to draw the arcball considering the manipulation state of the program. The 3D coordinates of the sphere is set to the object that is being manipulated as well as its rotation. Since we couldn’t complete the rotation code and translation fix code, the arcball scale is set to 1 as a stub.

```
  // draw arcball
  // ============

  // fix scale and you're good to go
  //g_arcballScale = getScreenToEyeScale(0, g_frustFovY, g_windowHeight);
  g_arcballScale = 1;

  switch(manipState){
  case 0:
    g_sphereRbt.setTranslation(Cvec3(0,0,0));
    break;
  case 1:
    g_sphereRbt.setTranslation(g_objRbt[0].getTranslation());
    g_sphereRbt.setRotation(g_objRbt[0].getRotation());
    break;
  case 2:
    g_sphereRbt.setTranslation(g_objRbt[1].getTranslation());
    g_sphereRbt.setRotation(g_objRbt[1].getRotation());
    break;
  }

  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  // draw wireframe

  // compute MVM, taking into account the dynamic radius
  float scale =  g_arcballScale * g_arcballScreenRadius;
  MVM = rigTFormToMatrix(inv(eyeRbt) * g_sphereRbt) * g_arcballScale;
  NMVM = normalMatrix(MVM);

  // send MVM and NMVM and colors
  sendModelViewNormalMatrix(curSS, MVM, NMVM);
  safe_glUniform3f(curSS.h_uColor, g_objectColors[2][0], g_objectColors[2][1], g_objectColors[2][2]);
  g_sphere->draw(curSS);

  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

To apply the desired arcball rotations to the object, the what it is rotated had to be changed. To differentiate the arcball rotation with the traditional rotation technique we used to use, two wrappers are implemented. First wrapper maintained the regular rotation code and since we couldn't complete the second part, the second wrapper stayed empty as follows:

The motion() callback is modified to replace big chunk of rotation code with wrappers where `m` is the motion `RigTForm`:

```
  if (g_mouseClickDown) {

   //noInterfaceRotation(m);
   arcBallRotation(m);

    glutPostRedisplay(); // we always redraw if we changed the scene
  }
```

As a final result of the assignment, we have successfully implemented and used RigTForms to translate and rotate objects. Successfully rendered the arcball in correct screen coordinates, however, failed to implement rotations, not to mention scaling when the object is translated in z axis.