cs839 Assignment 1

Ray Shulang Lei
200253624
Department of Computer Science
University of Regina

October 4, 2012

Abstract

This assignment is rendered by LaTex. assignment paper is at as1.pdf

1 Question 1

Let n be a 3 tuple vector, and given that it is along $V1$. It is trivial that we can imply:

$$n = \frac{V1}{[|V1|, |V1|, |V1|]}$$

where $|V1| = \sqrt{V1_x^2 + V1_y^2 + V1_z^2}$

Thus n is now known.

By the definition of cross product, denoted as $\times$ here, knowning that $V1$ and $V2$ is non-collinear, we can also derive:

$$u = \frac{V2 \times V1}{[|V2 \times V1|, |V2 \times V1|, |V2 \times V1|]}$$

Finally, it is also trivial that:

$$v = n \times u$$

2 Question 2

According to the requirement, we need a function that gets the new coordination U, V, N from two vectors.

First, assuming we have the function already. Thus giving it two vecotrs, our function will get the U, V, N from them.

```
<<src/q1_main.cpp>>=
#include <iostream>
#include <typeinfo>//debugging only
#include "util.h"

int main () {
  Vecotr V1;
  decltype(V1) V2;// V2 is of same type of V1

  V1 = {0,0,1000};
```

```
  V2 = {0,1,1};

  //call our function to get the uvn. auto will be replaced by the actual time by
  auto uvn = get_uvn(V1, V2);

  for (auto vecotr : uvn) {//for each Vecotr in uvn
    for (auto num : vecotr) {//for each number in Vecotr
      std::cout<<num<<',';
    }
    std::cout<<std::endl;
  }

  return 0;
}
@
```

I use a header file for typedefs and function declarations for more readable code.

```
<<src/util.h>>=
#ifndef VecotrS_HPP
#define VecotrS_HPP
#include <tr1/array>
typedef std::tr1::array<float, 3> Vecotr;
typedef std::tr1::array<Vecotr, 3> UVN;
UVN get_uvn(Vecotr V1, Vecotr V2);
float get_length(Vecotr);
Vecotr cross_product(Vecotr, Vecotr);
Vecotr normalize(Vecotr);
#endif
@
```

Finally, here is the function.

```
<<src/util.cpp>>=
#include "util.h"
#include <math.h>

//get u,v,n from two non-collinear vectors
UVN get_uvn(Vecotr V1, Vecotr V2) {
```

```
  //get n, which is just normalized V1
  Vecotr n = normalize(V1);

  //get u, which is normalized V2 x V1
  Vecotr u = normalize(cross_product(V2, V1));

  //get v, which is normalized n x u
  Vecotr v = normalize(cross_product(n, u));

  return {u,v,n};
}

//normalize a Vecotr
Vecotr normalize(Vecotr x) {
  return { x[0]/get_length(x),
           x[1]/get_length(x),
           x[2]/get_length(x) };
}

//calculates cross product of two Vecotrs
Vecotr cross_product(Vecotr x, Vecotr y) {
  return { x[1]*y[2] - x[2]*y[1],
           x[2]*y[0] - x[0]*y[2],
           x[0]*y[1] - x[1]*y[0]};
}

//calculates length of a Vecotr
float get_length(Vecotr x) {
  return sqrt(pow(x[0],2)+pow(x[1],2)+pow(x[2],2));
}
@
```

Furthermore, this is the command to link these files. Notice that I am using -std=c++11 flag to enable c++ 11 features. The output binary executable is bin/q1

```
<<compile_q1.sh>>=
clang++ -std=c++11 -o bin/q1 src/q1_main.cpp src/util.cpp
```

@

3 Question 3

3.1 part a

By definition of matrix multiplication,

$$T \times T^{-1} =$$

$$\begin{bmatrix} 1+0+0+0 & 0+0+0+0 & 0+0+0+0 & VRP_x+0+0+-VRP_x \\ 0+0+0+0 & 0+1+0+0 & 0+0+0+0 & 0+VRP_y+0+-VRP_y \\ 0+0+0+0 & 0+0+0+0 & 0+0+1+0 & 0+0+VRP_z+-VRP_z \\ 0+0+0+0 & 0+0+0+0 & 0+0+0+0 & 0+0+0+1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I$$

3.2 part b

Similarly, by definition of matrix multiplication,

$$R \times R^{-1} =$$

$$\begin{bmatrix} u_x^2+u_y^2+u_z^2 & u_x \times v_x + u_y \times v_y + u_z \times v_y & u_x \times n_x + u_y \times n_y + u_z \times n_y & 0 \\ v_x \times u_x + v_y \times u_y + v_z \times u_z & v_x^2+v_y^2+v_z^2 & v_x \times n_x + v_y \times n_y + v_z \times n_z & 0 \\ u_x \times n_x + u_y \times n_y + u_z \times n_z & n_x \times v_x + n_y \times v_y + n_z \times v_z & n_x^2+n_y^2+n_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} u \times u & u \times v & u \times n & 0 \\ v \times u & v \times v & v \times n & 0 \\ n \times u & n \times v & n \times n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With the fact that u, v, n are all unit vectors,

$$\implies u \times u = 1, v \times v = 1, n \times n = 1$$

$$\implies \begin{bmatrix} u \times u & u \times v & u \times n & 0 \\ v \times u & v \times v & v \times n & 0 \\ n \times u & n \times v & n \times n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & u \times v & u \times n & 0 \\ v \times u & 1 & v \times n & 0 \\ n \times u & n \times v & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With the fact that u, v, n are orthogonal to each other,

$$\implies u \times v = 0, v \times n = 0, u \times n = 0$$

$$\implies \begin{bmatrix} 1 & u \times v & u \times n & 0 \\ v \times u & 1 & v \times n & 0 \\ n \times u & n \times v & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = I$$

3.3 part c, d

Here I defined a series of functions to get to the final matrices that we need. get_M() is the M_{wc} and M_{wl} function. get_Mi() is the M_{cw} and M_{lw} function.

```
<<src/matrix.cpp>>=
#include "matrix.h"
#include "util.h"

//this is the world to view final matrix, which is Mwc, also Mwl
Matrix get_M(Point vrp, Point vpn, Point vup) {
  return mul(get_R(vrp, vpn, vup), get_T(vrp));
}

//this is the view to world final matrix, which is Mcw, also Mlw
Matrix get_Mi(Point vrp, Point vpn, Point vup) {
```

```
  return mul(get_Ti(vrp), get_Ri(vrp, vpn, vup));
}

//get transformation matrix
Matrix get_T(Point vrp) {
  Row r1 = {1, 0, 0, -vrp[0]};
  Row r2 = {0, 1, 0, -vrp[1]};
  Row r3 = {0, 0, 1, -vrp[2]};
  Row r4 = {0, 0, 0, 1};
  return {r1, r2, r3, r4};
}

//get inverse transformation matrix
Matrix get_Ti(Point vrp) {
  Row r1 = {1, 0, 0, vrp[0]};
  Row r2 = {0, 1, 0, vrp[1]};
  Row r3 = {0, 0, 1, vrp[2]};
  Row r4 = {0, 0, 0, 1};
  return {r1, r2, r3, r4};
}

//get rotation matrix
Matrix get_R(Point vrp, Point vpn, Point vup) {
  //first get the translation matrix from world to view
  auto mt = get_T(vrp);

  //second, translate the points to origin
  auto vpn_ = mul(mt, vpn);
  auto vup_ = mul(vup, mt);

  //now we can see vpn_ and vup_ as vectors. such that we can apply them to get_uv
  auto uvn = get_uvn(vup_, vpn_);
  //finally contruct our roation matrix using method 2 on class notes
  Row r1 = { uvn[0][0],uvn[0][1],uvn[0][2],0 };
  Row r2 = { uvn[1][0],uvn[1][1],uvn[1][2],0 };
  Row r3 = { uvn[2][0],uvn[2][1],uvn[2][2],0 };
  Row r4 = { 0, 0, 0, 1 };
  return { r1, r2, r3, r4 };
```

```
}

//get inverse rotation matrix
Matrix get_Ri(Point vrp, Point vpn, Point vup) {
  Matrix m = get_R(vrp, vpn, vup);
  Row r1 = { m[0][0], m[1][0], m[2][0], m[3][0] };
  Row r2 = { m[0][1], m[1][1], m[2][1], m[3][1] };
  Row r3 = { m[0][2], m[1][2], m[2][2], m[3][2] };
  Row r4 = { m[0][3], m[1][3], m[2][3], m[3][3] };
  return {r1,r2,r3,r4};
}

//matrix multiplication
Matrix mul(Matrix m, Matrix n) {
  Row r1 = {m[0][0]*n[0][0]+m[0][1]*n[1][0]+m[0][2]*n[2][0]+m[0][3]*n[3][0],
            m[0][0]*n[0][1]+m[0][1]*n[1][1]+m[0][2]*n[2][1]+m[0][3]*n[3][1],
            m[0][0]*n[0][2]+m[0][1]*n[1][2]+m[0][2]*n[2][2]+m[0][3]*n[3][2],
            m[0][0]*n[0][3]+m[0][1]*n[1][3]+m[0][2]*n[2][3]+m[0][3]*n[3][3]};
  Row r2 = {m[1][0]*n[0][0]+m[1][1]*n[1][0]+m[1][2]*n[2][0]+m[1][3]*n[3][0],
            m[1][0]*n[0][1]+m[1][1]*n[1][1]+m[1][2]*n[2][1]+m[1][3]*n[3][1],
            m[1][0]*n[0][2]+m[1][1]*n[1][2]+m[1][2]*n[2][2]+m[1][3]*n[3][2],
            m[1][0]*n[0][3]+m[1][1]*n[1][3]+m[1][2]*n[2][3]+m[1][3]*n[3][3]};
  Row r3 = {m[2][0]*n[0][0]+m[2][1]*n[1][0]+m[2][2]*n[2][0]+m[2][3]*n[3][0],
            m[2][0]*n[0][1]+m[2][1]*n[1][1]+m[2][2]*n[2][1]+m[2][3]*n[3][1],
            m[2][0]*n[0][2]+m[2][1]*n[1][2]+m[2][2]*n[2][2]+m[2][3]*n[3][2],
            m[2][0]*n[0][3]+m[2][1]*n[1][3]+m[2][2]*n[2][3]+m[2][3]*n[3][3]};
  Row r4 = {m[3][0]*n[0][0]+m[3][1]*n[1][0]+m[3][2]*n[2][0]+m[3][3]*n[3][0],
            m[3][0]*n[0][1]+m[3][1]*n[1][1]+m[3][2]*n[2][1]+m[3][3]*n[3][1],
            m[3][0]*n[0][2]+m[3][1]*n[1][2]+m[3][2]*n[2][2]+m[3][3]*n[3][2],
            m[3][0]*n[0][3]+m[3][1]*n[1][3]+m[3][2]*n[2][3]+m[3][3]*n[3][3]};
  return {r1,r2,r3,r4};
}

Point mul(Matrix m, Point x) {
  return mul(x, m);
}

Row mul(Row x, Matrix m) {
```

```
  return {x[0]*m[0][0]+x[1]*m[0][1]+x[2]*m[0][2]+x[3]*m[0][3],
          x[0]*m[1][0]+x[1]*m[1][1]+x[2]*m[1][2]+x[3]*m[1][3],
          x[0]*m[2][0]+x[1]*m[2][1]+x[2]*m[2][2]+x[3]*m[2][3],
          x[0]*m[3][0]+x[1]*m[3][1]+x[2]*m[3][2]+x[3]*m[2][3]};
}

Row mul(Matrix m, Row x) {
  return mul(x, m);
}

Point mul(Point x, Matrix m) {
  return {x[0]*m[0][0]+x[1]*m[0][1]+x[2]*m[0][2]+m[0][3],
          x[0]*m[1][0]+x[1]*m[1][1]+x[2]*m[1][2]+m[1][3],
          x[0]*m[2][0]+x[1]*m[2][1]+x[2]*m[2][2]+m[2][3]};
}

void pmatrix(std::string str, Matrix m) {
  std::cout<<str<<std::endl;
  for (auto row : m) {
    for (auto num : row) {
      std::cout<<std::setw (10);
      std::cout<<num;
    }
    std::cout<<std::endl;
  }
  std::cout<<std::endl;
}
@
```

Here I wrote a header file for main program to include. A matrix is simply 4 of 4-tuple vectors. So I defined my 4-tuple vector as Row type, and 4 Rows as Matrix type. A Point type is also defined to represent VRP, VPN, VUP, LRP, LPN and LUP.

```
<<src/matrix.h>>=
#ifndef MATRIX_H
#define MATRIX_H
#include <iostream>
#include <iomanip>
```

```
#include <string>
#include <tr1/array>
typedef std::tr1::array<float, 3> Point;
typedef std::tr1::array<float, 4> Row;
typedef std::tr1::array<Row, 4> Matrix;
Matrix get_T(Point);
Matrix get_Ti(Point);
Matrix get_R(Point, Point, Point);
Matrix get_Ri(Point, Point, Point);
Matrix get_M(Point, Point, Point);
Matrix get_Mi(Point, Point, Point);
Point mul(Point, Matrix);
Point mul(Matrix, Point);
Row mul(Row, Matrix);
Row mul(Matrix, Row);
Matrix mul(Matrix, Matrix);
void pmatrix(std::string, Matrix);
#endif
@
```

3.4 part f

```
<<src/q3_main.cpp>>=
#include "matrix.h"
int main(){
  //get camera test data points ready
  Point vrp = {6.0, 10.0, -5.0};
  Point vpn = {-6.0, -9.0, 5.0};
  Point vup = {0.0, 1.0, 0.0};

  //get matrix handy
  auto mt = get_T(vrp);
  auto mti = get_Ti(vrp);
  auto mr = get_R(vrp, vpn, vup);
  auto mri = get_Ri(vrp, vpn, vup);
  auto m_wc = get_M(vrp, vpn, vup);
  auto m_cw = get_Mi(vrp, vpn, vup);
```

```
//print results
pmatrix("translation matrix:", mt);
pmatrix("inverse translation matrix:", mti);
pmatrix("rotation matrix:", mr);
pmatrix("inverse rotation matrix:", mri);
pmatrix("world to camera matrix:", m_wc);
pmatrix("camera to world matrix:", m_cw);

//get light test data points ready
Point lrp = {-10.0, 10.0, 0.0};
Point lpn = {10.0, 9.0, 0.0};
Point lup = {0.0, 1.0, 0.0};

//get matrix handy
auto m_wl = get_M(lrp, lpn, lup);
auto m_lw = get_Mi(lrp, lpn, lup);

//print results
pmatrix("world to light matrix:", m_wl);
pmatrix("light to world matrix:", m_lw);

//print results
pmatrix("camera to light matrix:", mul(m_cw, m_wl));
pmatrix("light to camera matrix:", mul(m_lw, m_wc));

//now test the points
Row t1 = {0.0, 0.0, 0.0, 1.0};
Row t2 = {0.0, 1.0, 0.0, 1.0};
Row t3 = {1.0, 1.0, 1.0, 1.0};
Row t4 = {1.0, 1.0, 0.0, 1.0};

auto t1_ = mul(t1, m_wc);
auto t2_ = mul(t2, m_wc);
auto t3_ = mul(t3, m_wc);
auto t4_ = mul(t4, m_wc);
std::cout<<"test points at new camera coordinations:"<<std::endl;
std::cout<<t1_[0]<<", "<<t1_[1]<<", "<<t1_[2]<<", "<<t1_[3]<<std::endl;
```

```
  std::cout<<t2_[0]<<", "<<t2_[1]<<", "<<t2_[2]<<", "<<t2_[3]<<std::endl;
  std::cout<<t3_[0]<<", "<<t3_[1]<<", "<<t3_[2]<<", "<<t3_[3]<<std::endl;
  std::cout<<t4_[0]<<", "<<t4_[1]<<", "<<t4_[2]<<", "<<t4_[3]<<std::endl;

  auto t1__ = mul(t1, m_wl);
  auto t2__ = mul(t2, m_wl);
  auto t3__ = mul(t3, m_wl);
  auto t4__ = mul(t4, m_wl);
  std::cout<<"test points at new light coordinations:"<<std::endl;
  std::cout<<t1__[0]<<", "<<t1__[1]<<", "<<t1__[2]<<", "<<t1__[3]<<std::endl;
  std::cout<<t2__[0]<<", "<<t2__[1]<<", "<<t2__[2]<<", "<<t2__[3]<<std::endl;
  std::cout<<t3__[0]<<", "<<t3__[1]<<", "<<t3__[2]<<", "<<t3__[3]<<std::endl;
  std::cout<<t4__[0]<<", "<<t4__[1]<<", "<<t4__[2]<<", "<<t4__[3]<<std::endl;

  return 0;
}
@
```

And here is my linking and compiling commands.

```
<<compile_q3.sh>>=
clang++ -std=c++11 -o bin/q3 src/q3_main.cpp src/matrix.cpp src/util.cpp
@
```

3.5 execution results

And here are the results from executing bin/q3 on my machine: OS X 10.8

```
bin/q3
translation matrix:
         1         0         0        -6
         0         1         0       -10
         0         0         1         5
         0         0         0         1

inverse translation matrix:
         1         0         0         6
         0         1         0        10
```

```
         0          0          1         -5
         0          0          0          1

rotation matrix:
 -0.640184          0 -0.768221          0
  0.580209 -0.655422 -0.483508          0
 -0.503509 -0.755263  0.419591          0
         0          0          0          1

inverse rotation matrix:
 -0.640184  0.580209 -0.503509          0
         0 -0.655422 -0.755263          0
 -0.768221 -0.483508  0.419591          0
         0          0          0          1

world to camera matrix:
 -0.640184          0 -0.768221          0
  0.580209 -0.655422 -0.483508  0.655422
 -0.503509 -0.755263  0.419591   12.6716
         0          0          0          1

camera to world matrix:
 -0.640184  0.580209 -0.503509          6
         0 -0.655422 -0.755263         10
 -0.768221 -0.483508  0.419591         -5
         0          0          0          1

world to light matrix:
         0          0         -1          0
  0.668965  0.743294          0 -0.743294
  0.743294 -0.668965          0   14.1226
         0          0          0          1

light to world matrix:
         0  0.668965  0.743294        -10
         0  0.743294 -0.668965         10
        -1          0          0          0
         0          0          0          1
```

```
camera to light matrix:
 0.0138844  0.768096  0.640184  -1.54211
 -0.999837 0.0180734         0 -0.179101
-0.0115703  -0.64008  0.768221    1.2851
         0         0         0         1

light to camera matrix:
 0.0138844 -0.999837-0.0115703  -0.14279
  0.768096 0.0180734  -0.64008   2.01029
  0.640184         0  0.768221         0
         0         0         0         1

test points at new camera coordinations:
0, 0.655422, 12.6716, 12.6716
0, 5.96046e-07, 11.9164, 12.6716
-1.40841, 0.0967021, 11.8325, 12.6716
-0.640184, 0.58021, 11.4129, 12.6716
test points at new light coordinations:
0, -0.743294, 14.1226, 14.1226
0, -5.96046e-08, 13.4536, 14.1226
-1, 0.668965, 14.1969, 14.1226
0, 0.668965, 14.1969, 14.1226
```