DH2323 Lab3 Report

Shuang Qiu (shuangq@kth.se)

1. Drawing Points

We need to project the vertices from 3D to 2D. Before calculating the positions of the points, we need to transform the points from the world coordinate system to the system where the camera is at the origin. The function `VertexShader` does this job.

```
void VertexShader( const vec3& v, Pixel& p){
    vec3 p0;
    //transform the point from the world coordinate system
    //to the system where the camera is at the origin;
    p0 = (v - cameraPos) * R;

    //3D to 2D
   p.x = int(focalLength * p0.x / p0.z + SCREEN_WIDTH/2);
    p.y = int(focalLength * p0.y / p0.z + SCREEN_HEIGHT/2);
    p.zinv = 1.0f/p0.z;
}
```

Same as in lab 2, the position of the camera is set to (0, 0, -3) as shown in Figure 1.

2. Drawing Edges

After drawing all the vertices, we interpolate the points between them to draw the edges. We create a `Interpolate` function to calculate the position of the points between two vertices, and store them in an array.

```
void Interpolate( ivec2 a, ivec2 b, vector<ivec2>& result ){
    int N = result.size();
    ivec2 step = vec3(b-a) / float(max(N-1, 1));
    ivec2 current(a);
    for(int i=0; i<N; i++){
        result[i] = current;
        current += step;
```

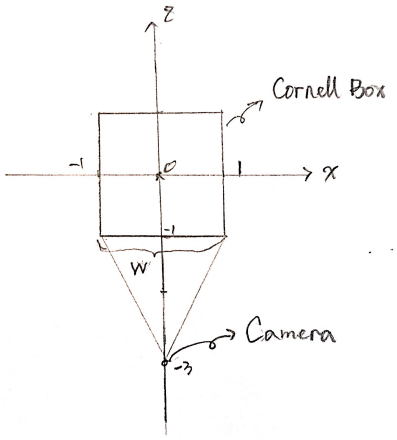


Figure 1: Camera position if focal length set to H

```
    }
}
```

Now we can interpolate each two vertices and draw the points in between, which will construct the edges.

```
void DrawLineSDL( SDL_Surface* surface, ivec2 a, ivec2 b, vec3 color){
    ivec2 delta = glm::abs(a - b);
    int pixels = glm::max( delta.x, delta.y) + 1;
    vector<ivec2> line(pixels);
    Interpolate(a, b, line);

    if( SDL_MUSTLOCK(screen) )
        SDL_LockSurface(screen);

        // drawing line
        for(int i = 0; i < pixels; i++){
            PutPixelSDL(screen, line[i].x, line[i].y, color);
        }

    if ( SDL_MUSTLOCK(screen) )
        SDL_UnlockSurface(screen);
}
```

Then we need to interpolate between the edges to get the value of every pixel.

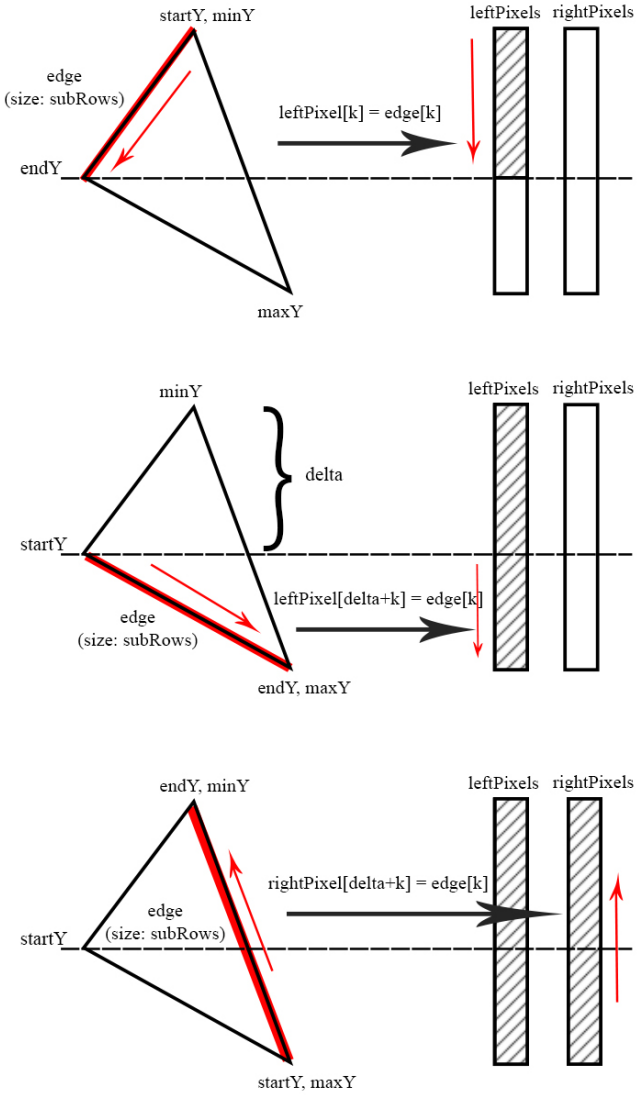
Moving Camera by Mouse: I implemented mouse control using the following method.

```
//     rotate by mouse
    int curX, curY;
    int dx;

    if(SDL_GetMouseState(NULL, NULL) & SDL_BUTTON(SDL_BUTTON_LEFT)){
        SDL_GetMouseState( &curX, &curY);
        dx = curX - oldX;

        if(dx > 0){
            yaw += 0.05f;
            rotateY();

        }else if(dx < 0){
            yaw -= 0.05f;
            rotateY();
        }
        oldX = curX;
    }
```

3. Filled Triangles

Following the pseudo-code, I populate the `ComputePolygonRow()` function. The general idea is to break the left side (or right side) of a triangle to two parts, and use an array `edge[subRows]` to store the pixels of each part. Then copy the value in `edge` to the corresponding position in `leftPixels` / `rightPixels`.

It is worth noting that the order to interpolate the value is important. If the direction is positive (endY >= startY), the value should be interpolated from the current vertex to the next one; otherwise if the direction is negative (endY < startY), the value should be interpolated from the next vertex to the current one.



Related code:

```
void ComputePolygonRows( const vector<Pixel>& vertexPixels,
   vector<Pixel>& leftPixels, vector<Pixel>& rightPixels ){
    // 1. Find max and min y-value of the polygon
    // and compute the number of rows it occupies
    // ...

    // 2. Resize leftPixels and rightPixels so that
    // they have an element for each row
    // ...

    // 3. Initialize the x-coordinates in leftPixels to
    // some really large value and the x-coordinates in
    // rightPixels to some really small value
    // ...

    // 4. Loop through all edges of the polygon and use linear
    // interpolation to find the x-coordinate for each row it
    // occupies. Update the corresponding values in rightPixels
    // and leftPixels
    vector<Pixel> edge;

    for(int i=0; i<V; i++){
        int j = (i + 1) % V;    //the next vertex
        int startY, endY;
        bool direction = false;

        startY = vertexPixels[i].y;
        endY = vertexPixels[j].y;
        if(startY <= endY){
            direction = true;
        }

        int subRows = glm::abs(endY - startY) + 1;
        edge.resize(subRows);

        if(direction){
            Interpolate(vertexPixels[i], vertexPixels[j], edge);
        }else{
            Interpolate(vertexPixels[j], vertexPixels[i], edge);
        }

        int delta = direction ? glm::abs(startY - minY) :
        glm::abs(endY - minY) ;

        for(int k=0; k < subRows; k++){
            if(leftPixels[delta + k].x > edge[k].x){
```

```
                leftPixels[delta + k] = edge[k];
            }
            if(rightPixels[delta + k].x < edge[k].x){
                rightPixels[delta + k] = edge[k];
            }
        }
    }
}
```

4. Depth Buffer

Since in perspective projection the depth along the edge will not vary linearly, but 1/z will, here we use a new struct for the pixels:

```
struct Pixel{
    int x;
    int y;
    float zinv;
};
```

Therefore we also need to change the related functions. For example, `Interpolate()`:

```
void Interpolate( Pixel a, Pixel b, vector<Pixel>& result ){
    int N = result.size();
    float stepX = (b.x - a.x) / float(max(N-1, 1));
    float stepY = (b.y - a.y) / float(max(N-1, 1));
    float stepZ = (b.zinv - a.zinv) / float(max(N-1, 1));

    Pixel current(a);

    for(int i=0; i<N; i++){
        current.x = stepX * i + a.x;
        current.y = stepY * i + a.y;
        current.zinv = stepZ * i + a.zinv;
        result[i] = current;
    }
}
```

When implementing this function, I met a problem. I used `current.x += stepX` instead. Because `current.x` is an int, so being incremented by a small float, its value will not change.

5. Illumination

There are two ways to implement illumination: **Per Vertex Illumination** and **Per Pixel Illumination**.

1. Per Vertex Illumination: compute the illumination value for every vertex, and then interpolate the values in between.
2. Per Pixel Illumination: interpolate the 3D position for every pixel, then use the 3D position to calculate the illumination.

6. Final Output

The output is shown in Figure 3.

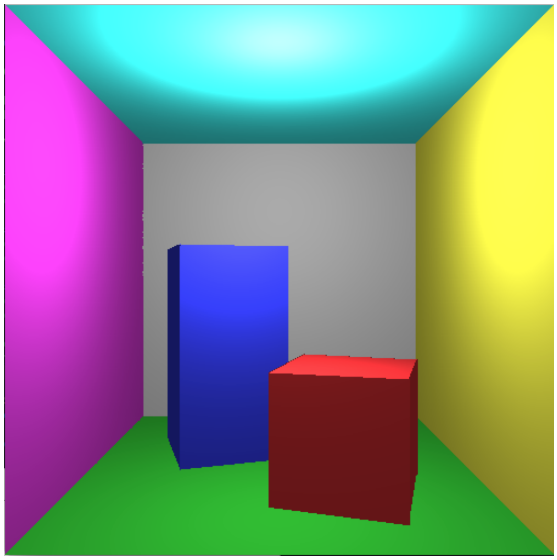


Figure 2: Final output