COMS30115 - Coursework 2

Rasterisation

Carl Henrik Ek & Magnus Burenius

February 26, 2019

Raytracing is the simplest way of computing a 2D image of a 3D scene. It can be used to simulate all kinds phenomena, much more than we implemented in the second lab. Its only disadvantage is speed why it is therefore typically not used for real-time visualization. For those scenarios another method called rasterization is often used. Although rasterization is typically faster than raytracing itis not as easy to implement and it cannot be used to simulate all illumination phenomena. It is for instance very difficult to simulate multiple bounces of light with it. In this lab you will implement a rasterizer. The lab consists of three parts. In the first part you will explore:

- Perspective projection of 3D points by a pinhole camera.
- Drawing 3D objects modeled by triangular surfaces by first projecting the vertices of the triangles to the 2D image plane.
- Drawing the edges of each triangle as lines in 2D, using linear interpolation.

The result of this can be seen in the top image in Figure 1. You will then add:

- Drawing of filled triangles.
- A depth buffer to check visibility for each pixel.

The result of this is very similar to the first part of lab 2, as can be seen in the middle image of figure 1, but computed much faster. In the last part you will extend the program by also implementing:

- Light sources.
- Interpolation of arbitrary quantities across the triangle, although you will focus on light.
- Per vertex and per pixel illumination, implemented by vertex and pixel shaders.

The final result can be seen in the bottom image of Figure 1.

1 Transformations

1.1 Perspective Projection of Points from 3D to 2D

In the first lab you made a starfield by simulating a pinhole camera and its projection of 3D points to an image. In the second lab you did a raytracer by also simulating a pinhole camera, but you did not project points. Instead you sent out rays for each pixel of the camera image. These are the two main approaches that can be used to make a 2D image of a 3D world. In this lab we will go back to the first approach and use projection instead of raytracing.

Assume we have a pinhole camera positioned at the origin looking along the z-axis of a 3D coordinate system. Let the x-axis point to the right and let the y-axis point downwards. Let f be the focal length of the camera, i.e. the distance from the camera position to the image plane, measured in pixel units. Let the x-

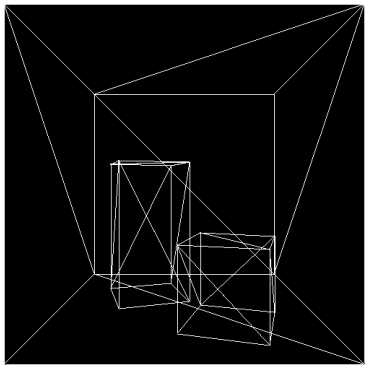
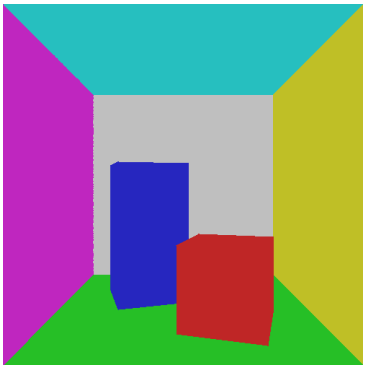
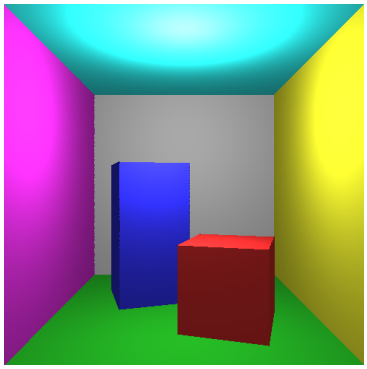

Table 1: The output from this lab.

and y-axis of the 2D camera image be the same as for the 3D world, i.e. the origin will be in the "middle" of the image. Then the projection of an arbitrary 3D point (X, Y, Z) to a 2D point (x, y) in the camera image can be written as:

$$x = f\frac{X}{Z} \tag{1}$$

$$y = f\frac{Y}{Z} \tag{2}$$

This relation can be derived by looking at two triangles that have the same angles. However, when working with computers it is common to have the origin of the image in the top left corner. If the image has width W and height H the projection can then be written:

$$x = f\frac{X}{Z} + \frac{W}{2} \tag{3}$$

$$y = f\frac{Y}{Z} + \frac{H}{2} \tag{4}$$

1.2 Translation

Assume the camera is not fixed at the origin but can move around freely. Let C be the position of the camera. Then to project a 3D point using equation and 4 we first need to transform the point P from the world coordinate system to the system where the camera is at the origin. We can do this by subtracting the position of the camera:

$$P' = P - C \tag{5}$$

P' is then the position of the point in a coordinate system centered at the camera.

1.3 Rotation

Now we assume that the camera position is fixed at the origin again but that it can rotate around it. We then have a fixed world coordinate system and another coordinate system that rotates with the camera. The transformation of a position vector P from the world coordinate system to the system of the camera can then be expressed using a 3×3 matrix R and vector-matrix multiplication:

$$\underbrace{P'}_{3\times1} = \underbrace{R}_{3\times3}\underbrace{P}_{3\times1} \tag{6}$$

where P and P' are row vectors. We say that the matrix R that can be used to perform this transformation is the rotation matrix of the camera.

1.4 Translation & Rotation

Finally we assume that the camera can both rotate and translate. Then we can perform the transformation of a point from world coordinate system to camera coordinate system in two steps:

- Transform it from world space to a coordinate system that is centered at the camera using equation 5
- Transform it from the coordinate system that is centered at the camera to a system that is also rotated as the camera using equation 6.

The total transformation then becomes,

$$P' = R \cdot (P - C) \tag{7}$$

where the order of the operation implies that we first translate and then rotate.

1.5 Homogenous Coordinates

Keeping the rotations and transformations as separate operations can be rather tricky as you have to remember the order of the transformations. A better way of working is to do everything in projective or homogenous coordinates instead. The above transformation could then be written as,

$$\begin{bmatrix} 0 & 0 & 0 & c_x \\ 0 & 0 & 0 & c_y \\ 0 & 0 & 0 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{11} & R_{12} & R_{13} & 0 \\ R_{11} & R_{12} & R_{13} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 & -c_x \\ 0 & 0 & 0 & -c_y \\ 0 & 0 & 0 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

where all transformations are now written as multiplications. I recommend that you work with homogenous coordinates througout as it makes life quite a bit easier. Importantly if you are interested in implementing clipping for the extensions you will have **a lot** of benefit from using this representation. There are many ways of specifying the rotation matrix, as a set of angles, or as a single angle around a specific vector. The Wikipedia article URL has quite a good description of the latter. It doesn't matter how you do it but write a simple function that creates a transformation matrix.

Code

```
void TransformationMatrix(glm:mat4x4 M, ...)
```

2 Drawing Points

First just try to project and plot the vertices of the scene, similar to how you did for the starfield in the first lab. The Draw-function from the template should be altered to something looking like this,

Code

```
void Draw()
{
  /* Clear buffer */
  memset(screen->buffer, 0, screen->height*screen->width*sizeof(uint32_t));

  for( uint32_t i=0; i<triangles.size(); ++i )
    {
      vector<vec4> vertices(3);

      vertices[0] = triangles[i].v0;
      vertices[1] = triangles[i].v1;
      vertices[2] = triangles[i].v2;

      for(int v=0; v<3; ++v)
        {
          ivec2 projPos;
          VertexShader( vertices[v], projPos );
          vec3 color(1,1,1);
          PutPixelSDL( screen, projPos.x, projPos.y, color );
        }
    }
}
```

The function loops through all triangles and all vertices of the triangles and calls the function VertexShader on each. You have to implement this function:

Code

```
void VertexShader( const vec4& v, ivec2& p );
```

It should take the 4D position of a vertex v and compute its 2D image position and store it in the given 2D integer vector p. `glm::ivec2` is a data type for 2D integer vectors, i.e. a pixel position will be represented by two integers. Thus it should handle the translation and rotation of the camera as well as the perspective projection. If you want you can wait with implementing the rotation and also the motion of the camera in the Update-function. These things will be easier to debug later when you will see something more than just some sparse points on the screen. You can start by just having a fixed position of the camera represented by the variable:

Code

```
vec4 cameraPos( 0, 0, -3.001,1 );
```

If you got this working you should see something similar to figure 1, i.e. points at the corners/vertices of the room and the two boxes.

3 Drawing Edges

To make the visualization slightly more interesting we will try to also draw the edges of the triangles, instead of just the vertices. We can do this by drawing lines in 2D between the projected vertices of the triangle. To draw lines in 2D you can use a function that does linear interpolation similar to what you wrote for the first lab. Instead of interpolating pixel colors represented by glm::vec3 we will interpolate pixel positions represented by `glm::ivec2`. In this lab you will later extend the function to interpolate lots of different

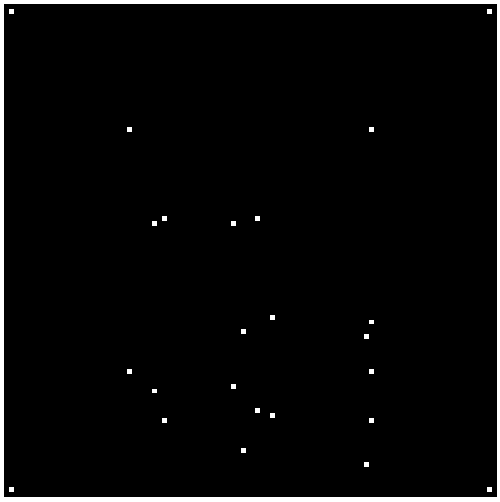

Figure 1: Drawing points projected by a pinhole camera.

values and it is therefore convenient if the interpolation is implemented in a simple but efficient way. As you might remember from the first lab this is a good way to do it:

Code

```
void Interpolate( ivec2 a, ivec2 b, vector<ivec2>& result )
{
    int N = result.size();
    vec2 step = vec2(b-a) / float(max(N-1,1));
    vec2 current( a );
    for( int i=0; i<N; ++i )
    {
        result[i] = current;
        current += step;
    }
}
```

Just as in the first lab the Interpolation-function takes the start value a and the end value b and fills the `std::vector` result with values linearly interpolated in between. The size of the result vector should have been set before calling this function, as it determines the number of discretization steps of the interpolation. For example to create a `std::vector` containing the values $(4,2),(3,4),(2,6),(1,8)$ we can write:

Code

```
vector<ivec2> result(4);
ivec2 a(4,2);
ivec2 b(1,8);
Interpolate( a, b, result );
```

Make sure that you understand how the interpolation code works as you will use it a lot and also extend it to work for other quantities than 2D positions. We are going to do a lot of interpolation with different types in this lab so it might be good to write a general method using `C++` templates[1]. Templates can be really useful you are actually using them for the `glm` types already, the downside is that debugging becomes a bit of a chore sometimes and because you have abstracted away the type you have less scope for optimisation.

[1] `http://en.cppreference.com/w/cpp/language/function_template`

It might be good to know that although this is a simple way to perform interpolation of integers it is not as fast as Bresenham's line algorithm. However, Breshenham is less intuitive and for the basic part of the lab nothing you need to implement.

To draw a line in 2D we first need to know how many pixels the line should consist of. We do not want any holes in the line. Depending on whether the line is mostly horizontal or vertical we will use one pixel per column or one pixel per row. If a and b are the start and end of the line segment we can then compute the number of pixels to draw as:

Code

```
ivec2 delta = glm::abs( a - b );
int pixels = glm::max( delta.x, delta.y ) + 1;
```

You can then get the pixel positions of the line by calling the Interpolation function:

Code

```
vector<ivec2> line( pixels );
Interpolate( a, b, line );
```

When we have these we can loop through all of them and call PutPixelSDL for these pixel positions to draw the line. Write a function that does this:

Code

```
void DrawLineSDL( SDL_Surface* surface, ivec2 a, ivec2 b, vec3 color );
```

Before applying it to draw the edges of the projected triangles, try to draw lines between some arbitrary image points. When you got that working add another function to draw the edges of a triangle:

Code

```
void DrawPolygonEdges( const vector<vec4>& vertices )
{
    int V = vertices.size();

    // Transform each vertex from 3D world position to 2D image position:
    vector<ivec2> projectedVertices( V );
    for( int i=0; i<V; ++i )
    {
        VertexShader( vertices[i], projectedVertices[i] );
    }

    // Loop over all vertices and draw the edge from it to the next vertex:
    for( int i=0; i<V; ++i )
    {
        int j = (i+1)%V; // The next vertex
        vec3 color( 1, 1, 1 );
        DrawLineSDL( screen, projectedVertices[i], projectedVertices[j], color );
    }
}
```

It takes the vertices of the triangle, or in fact any polygon, and project them to the image and then draw a line for each edge, using DrawLineSDL. The % operator gives the remainder after division of two integers. We use it to make the next index after the last index wrap around to the first index. The operator works

like this:

$$\begin{aligned} 0\%3 &= 0 \\ 1\%3 &= 1 \\ 2\%3 &= 2 \\ 3\%3 &= 0 \\ 4\%3 &= 1 \\ 5\%3 &= 2 \\ 6\%3 &= 0 \end{aligned} \tag{8}$$

You can then use DrawPolygonEdges in the Draw-function like this:

Code

```
void Draw()
{
  /* Clear buffer */
  memset(screen->buffer, 0, screen->height*screen->width*sizeof(uint32_t));

  for( uint32_t i=0; i<triangles.size(); ++i )
    {
      vector<vec4> vertices(3);

      vertices[0] = triangles[i].v0;
      vertices[1] = triangles[i].v1;
      vertices[2] = triangles[i].v2;

      DrawPolygonEdges( vertices );
    }
}
```

This should give you an image of a wire frame scene like in figure 2. Now when you have a better visualization of the scene you should also implement motion of the camera if you have not done that yet. Have variables for the position of the camera and its rotation.

Code

```
glm::vec4 cameraPos( 0, 0, -3.001, 1 );
glm::mat4 R;
float yaw = 0;        // Yaw angle controlling camera rotation around y-axis
```

Update these in the Update-function when the user presses the arrow keys just as you did in lab 2 and make sure that the VertexShader-function handles both the translation and rotation of the camera before it projects a point from 3D to 2D. It should be possible to rotate the camera around the y-axis. This type of rotation is called yaw.

If you want you can also add the possibility to rotate the camera up and down pitch rotation), but that is not mandatory. Another thing that you can add if you want is control of the camera rotation by the mouse instead of the keyboard. This is also not mandatory. You can get access the relative motion of the mouse by calling `SDL_GetRelativeMouseState`:

Code

```
int dx;
int dy;
SDL_GetRelativeMouseState( &dx, &dy );
```

This function can also be used to see which mouse buttons that are pressed. You can read more about it in the SDL documentaion.

When you move around with the camera you might notice that there are problems if the vertices are behind the camera. This is actually a bit tricky to fix and nothing you need to do for this lab. The solution to the problem is to perform clipping of the triangles before they are drawn. Clipping is a topic that could be studied in the optional project.

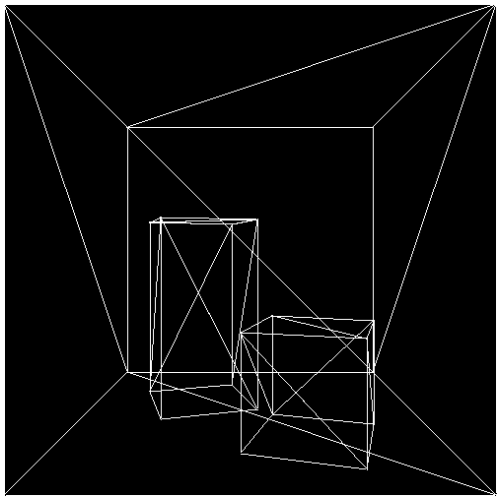

Figure 2: Drawing edges.

4 Filled Triangles

By just drawing the edges of the triangles we get some feeling for the 3D structure of the scene which might be good for some engineering visualizations. Nevertheless, it would be nice if we could also draw solid surfaces, i.e. fill the triangles with color. This turns out to be a bit more involved. The main idea is to draw the triangle row by row. We have an array that stores the start position and another array that stores the end position of the triangle for each row:

Code

```
vector<ivec2> leftPixels( ROWS );
vector<ivec2> rightPixels( ROWS );
```

Assume we have somehow filled these arrays with values, then it is simple to loop through them and draw each row from the start to the end. The tricky part is to compute the arrays in the first place.

As an example consider a triangle which has the projected vertices: $(10, 20)$, $(30, 10)$, $(20, 40)$, it should be drawn as $40 - 10 + 1 = 31$ rows. The arrays for the left and right positions should then have 31 elements each. Corresponding elements should have the same y-coordinate but different x-coordinates: the left and right of that row.

One way to fill these arrays with values representing the polygon is to first initialize the start arrays with really big values and the end array with really small values:

Code

```
for( int i=0; i<ROWS; ++i )
{
    leftPixels[i].x  = +numeric_limits<int>::max();
    rightPixels[i].x = -numeric_limits<int>::max();
}
```

We then loop through the edges of the polygon and compute the pixels corresponding to its line. Then for each y-coordinate of this line we check the corresponding elements in the left and right arrays. If the current x-value in the left array at that place is larger than the x-value of the line we replace it. If the current x-value in the right array at that place is smaller than the x-value of the line we replace it.

After we have looped through all edges we then have the smallest x-value for each row in the left array and the largest value for each row in the right array. Write a function that does this:

Code

```
void ComputePolygonRows(
    const vector<ivec2>& vertexPixels,
    vector<ivec2>& leftPixels,
    vector<ivec2>& 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.
}
```

It should take a vector with the projected position of the three vertices and compute the start and end image position of each row of the triangle. In fact, we can use this function not only to draw triangles but any convex polygon. You can test that your function produces sensible output by writing something like:

Code

```
vector<ivec2> vertexPixels(3);
vertexPixels[0] = ivec2(10, 5);
vertexPixels[1] = ivec2( 5,10);
vertexPixels[2] = ivec2(15,15);
vector<ivec2> leftPixels;
vector<ivec2> rightPixels;
ComputePolygonRows( vertexPixels, leftPixels, rightPixels );
for( int row=0; row<leftPixels.size(); ++row )
{
    cout << "Start: ("
         << leftPixels[row].x << ","
         << leftPixels[row].y << "). "
         << "End: ("
         << rightPixels[row].x << ","
         << rightPixels[row].y << "). " << endl;
}
```

This should give the output:

Code

```
Start: (10,5). End: (10,5).
Start: (9,6). End: (10,6).
Start: (8,7). End: (11,7).
Start: (7,8). End: (11,8).
Start: (6,9). End: (12,9).
Start: (5,10). End: (12,10).
Start: (7,11). End: (13,11).
Start: (9,12). End: (13,12).
Start: (11,13). End: (14,13).
Start: (13,14). End: (14,14).
Start: (15,15). End: (15,15).
```

When you got this working you can write a function that draws the computed rows:

Code

```
void DrawRows( const vector<ivec2>& leftPixels,
               const vector<ivec2>& rightPixels );
```

This function should call PutPixelSDL for each pixel between the start and end for each row. Finally write a function that projects the vertices and calls the other two functions:

Code

```
void DrawPolygon( const vector<vec4>& vertices )
{
    int V = vertices.size();

    vector<ivec2> vertexPixels( V );
    for( int i=0; i<V; ++i )
        VertexShader( vertices[i], vertexPixels[i] );

    vector<ivec2> leftPixels;
    vector<ivec2> rightPixels;
    ComputePolygonRows( vertexPixels, leftPixels, rightPixels );
    DrawPolygonRows( leftPixels, rightPixels );
}
```

Then call DrawPolygon in the Draw-function instead of DrawPolygonEdges. To signal what color to draw you can create a new variable:

Code

```
vec3 currentColor;
```

which you use in DrawPolygonRows when you call PutPixelSDL. We set this color to the color of the current triangle when we loop over all triangles and call DrawPolygon in the Draw-function:

Code

```
  void Draw()
  {
    /* Clear buffer */
    memset(screen->buffer, 0, screen->height*screen->width*sizeof(uint32_t));

    for( uint32_t i=0; i<triangles.size(); ++i )
      {
        vector<vec4> vertices(3);

        vertices[0] = triangles[i].v0;
        vertices[1] = triangles[i].v1;
        vertices[2] = triangles[i].v2;

        DrawPolygon( vertices );
      }
  }
```

You should then get the result seen in figure 3. When you manage to get that you are done with most of the coding for this lab. The remaining stuff does not require as much coding but greatly improves the image. Notice how the blue box is drawn on top of the red since we have not added any mechanism to deal with occlusion yet. Compare the speed of the program with the raytracer. How much faster is it for this scene?

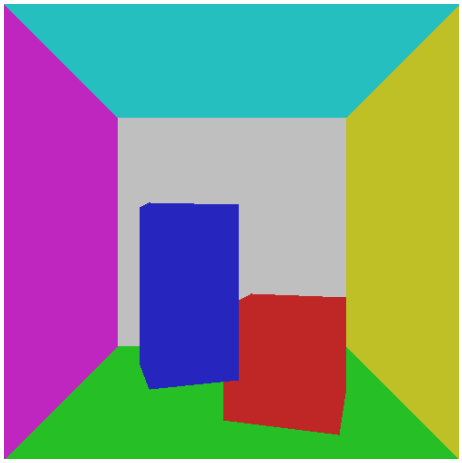

Figure 3: Filled triangles. Notice how the blue box is drawn on top of the red.

5 Depth Buffer

You have now implemented the core of a rasterizer. However, a problem with the current implementation is that the triangles might be drawn on top of each other, in arbitrary order. If multiple triangles occupy the same pixel we would like the one closest to the camera to be visible. In the raytracer we managed this by treating all pixels independently. For each pixel we followed a ray and looked at the closest intersection for that ray.

In a rasterizer we try to speed things up by not treating every pixel independently. We just treat every triangle independently. The standard way to handle this occlusion problem in a rasterizer is to use a so called depth buffer. That is an image storing a depth value for each pixel instead of a color value. For each pixel we thus store the depth of the closest surface point that has been drawn at that position so far. When we draw a new pixel we can then check if its depth is smaller than the one currently stored in the depth buffer. Only then do we draw it and also replace the value in the depth buffer.

To do this we need to keep track of the depth of each pixel in the image. We can do this by interpolating the depth over the triangle when we draw it. First we compute the depth of the vertices, in the coordinate system of the camera. Then we interpolate the depth over each edge of the triangle. Finally, we interpolate the depth over each row when we draw. This is similar to the bilinear interpolation we used in the first lab. First we interpolate the left and right edge from top to bottom. Secondly, we interpolate each row from left to right.

However, the perspective projection makes the interpolation a bit more involved. Assume z_1 is the depth of one of the vertices and z_2 is the depth of another vertex. Then in the 3D world the depth along the edge will vary linearly between z_1 and z_2, since the surface is planar. However, in the image the depth along the edge will not vary linearly! Instead it turns out that $1/z$ will vary linearly, due to the perspective projection of the camera. We can thus compute this quantity for each vertex and then interpolate it linearly across the projected 2D triangle.

After interpolation we could take the inverse to get back to the depth z. However, we do not really need the depth z. The inverse $1/z$ which we already have is good enough. We can then store $1/z$ in the depth buffer for each pixel that is drawn. A new pixel should then be drawn if its $1/z$ is larger than the one already stored in the depth buffer. Then its depth z will be smaller and it is closer to the camera. To implement a depth buffer we create a 2D array with the same size as the camera image:

Code

```
float depthBuffer[SCREEN_HEIGHT][SCREEN_WIDTH];
```

In it we will store the inverse depth $1/z$ for each pixel. Since we now also need to interpolate the depth

over the triangle, not just 2D position, we create a new struct to hold the information needed for each pixel:

Code

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

Previously we just interpolated the position between the vertices when we a triangle. Now we also want to interpolate the inverse depth. Thus, you need to implement an Interpolation function that interpolates our Pixel-struct linearly instead of `glm::ivec2`:

Code

```
void Interpolate( Pixel a, Pixel b, vector<Pixel>& result );
```

After you have done this you also need to change `ComputePoloygonRows`, `DrawPolygonRows`, `VertexShader` and `DrawPolygon` to handle Pixel instead of `glm::ivec2`:

Code

```
void ComputePolygonRows(
    const vector<Pixel>& vertexPixels,
    vector<Pixel>& leftPixels,
    vector<Pixel>& rightPixels );

void DrawPolygonRows(
    const vector<Pixel>& leftPixels,
    const vector<Pixel>& rightPixels );

void VertexShader( const vec4& v, Pixel& p )
```

Thus, in VertexShader you should now compute the inverse depth zinv for the vertex in the coordinate system of the camera, before you compute its projected image position (x, y).

If you do all this you will have access to zinv for every pixel that you draw in DrawPolygonRows. Before drawing a new pixel you can then check if it is in front of the one that is currently drawn there by comparing with the value of depthBuffer at that pixel. If it is in front you can draw it and update the value in the depth buffer. Remember that you also need to clear the depth buffer in the beginning of the Draw-function before you start drawing. You do this by setting all pixels to represent infinite depths, i.e. zero inverse depths:

Code

```
for( int y=0; y<SCREEN_HEIGHT; ++y )
    for( int x=0; x<SCREEN_WIDTH; ++x )
        depthBuffer[y][x] = 0;
```

You should then get surfaces that correctly occlude each other like in figure 4.

6 Illumination

Your rasterizer now renders the same image as the first version of the raytracer, but hopefully much faster. We will now continue and also add illumination. When computing the illumination in a rasterizer we have

Figure 4: Filled triangles. Notice how the blue box is no longer drawn on top of the red.

two choices. We could do the computations for every vertex and then just interpolate the result or we could do the computations for every pixel. The first approach will be faster but less accurate. Either way it is convenient to have one functions that handles whatever computations we want to do per vertex:

Code

```
void VertexShader( const vec4& v, Pixel& p );
```

which we already have, and another function which handles whatever computations we want to do per pixel:

Code

```
void PixelShader( const Pixel& p );
```

This function should be called for every interpolated pixel in DrawPolygonRows and handle the drawing. In the current version of our razterizer it might look like:

Code

```
void PixelShader( const Pixel& p )
{
    int x = p.x;
    int y = p.y;
    if( p.zinv > depthBuffer[y][x] )
    {
        depthBuffer[y][x] = f.zinv;
        PutPixelSDL( screen, x, y, currentColor );
    }
}
```

Currently VertexShader takes a `glm::vec3` and computes a Pixel. You have probably written something like:

Code

```
void VertexShader( const vec4& v, Pixel& p )
{
    vec4 pos = T*v;
    p.zinv = 1/pos.z;
    p.x = int(focalLength * pos.x * p.zinv) + SCREEN_WIDTH/2;
    p.y = int(focalLength * pos.y * p.zinv) + SCREEN_HEIGHT/2;
}
```

To make our rasterizer a bit more general and abstract we can create a new type which describes a general vertex:

Code

```
struct Vertex
{
    vec4 position;
};
```

For now we just store the position of the vertex since that is all we use at the moment. However, in general we could also assign many other quantities to the vertex. We will soon do this to handle illumination but first make your program handle this simple Vertex-type by updating: Draw, DrawPolygon and VertexShader.

The flow of your general rasterizer can now be described as:

- VertexShader is called for each Vertex and computes a Pixel.
- These Pixels are interpolated in the image between the vertices. First vertically along the edges and then horizontally across the polygon.
- Each such interpolated Pixel is sent to PixelShader which determines the final color of the image at that position.

Most of the code you have written is to perform the second step. There is not that much code in VertexShader and PixelShader. The good news is that to render more sofisticated images, e.g. with illumination and texture mapping, you do not need to add things to the second step. You just need to alter `VertexShader`, `PixelShader` and the interpolation function. In fact, if you would use a rasterization library like `OpenGL` or `Micro$oft's DirectX`, you would more or less only need to write the first and third step. These libraries completely handle the cumbersome second step for you.

6.1 Per Vertex Illumination

We will first try to implement the illumination computations for every vertex and then interpolate these values accross the polygon, similar to how we interpolated zinv. In lab 2 you learned that the direct illumination from an omni light source to a surface point can be written as:

$$D = \frac{P \max(\hat{r} \cdot \hat{n}, 0)}{4\pi r^2} \tag{9}$$

where D is the power of the incoming direct light per surface area. P is the power of the light source, r is a vector from the surface point to the light source and $\hat{n}$ is the normal of the surface. For ideally diffuse surfaces with reflectance ρ the total reflected light is then:

$$R = \rho * (D + N) \tag{10}$$

where N is the incoming indirect illumination reflected from other surfaces. We approximated N as a constant term. To implement this illumination model we will store the parameters of the light source as variables just as in lab 2:

Code

```
vec4 lightPos(0,-0.5,-0.7);
vec3 lightPower = 1.1f*vec3( 1, 1, 1 );
vec3 indirectLightPowerPerArea = 0.5f*vec3( 1, 1, 1 );
```

We would initially like to compute the illumination for every vertex in VertexShader. We then need to evaluate equation 9 and 10 for every vertex. Besides the light variables we then need to know the position, normal and reflectance at the vertex. It is therefore convenient to add this information to our Vertex-struct:

Code

```
struct Vertex
{
    vec4 position;
    vec4 normal;
    vec2 reflectance;
};
```

Make sure to set this information for every vertex in the Draw-function. After you have done this you will have access to this in VertexShader. You can then compute the illumination, but you also need to store it for the resulting output Pixel of VertexShader. You therefore need to add another quantity to your Pixel-struct to store this:

Code

```
struct Pixel
{
    int x;
    int y;
    float zinv;
    vec3 illumination;
};
```

Now the illumination gets computed for every Vertex in VertexShader. We then want to interpolate this value over the polygon before we use it in PixelShader. To do this you need to modifify the Interpolation-function so that it also interpolates the illumination quantity in Pixel. After you have done this you can simply access the illumination in PixelShader and use it when you draw the pixel:

Code

```
void PixelShader( const Pixel& p )
{
    int x = p.x;
    int y = p.y;
    if( p.zinv > depthBuffer[y][x] )
    {
        depthBuffer[y][x] = f.zinv;
        PutPixelSDL( screen, x, y, p.illumination );
    }
}
```

This should give you the result seen in figure 5. As you can see the result is similar to the raytracer with illumination but not exactly the same. Since we are interpolating the illumination over the surfaces we get less detail, but gain speed.

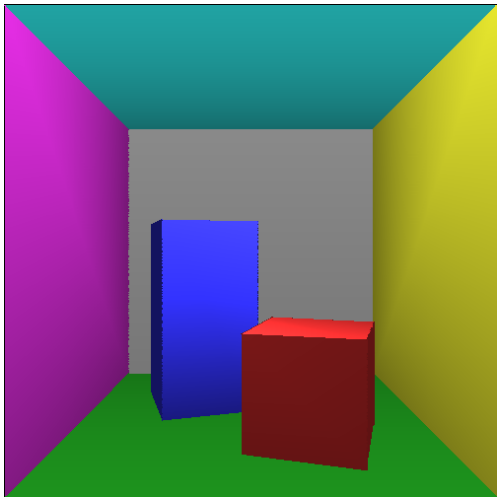

Figure 5: Illumination computed for vertices and interpolated in between.

6.2 Per Pixel Illumination

To get a more detailed illumination we will now compute unique values for every pixel. We do this by evaluating equation 9 and 10 in PixelShader instead of VertexShader. We then need to have access to the 3D position that the pixel represents as well as its normal and reflectance. Since we model the normal and reflectance as constant over each triangle we do not need to interpolate them. Instead we can store these values for the triangle that is currently drawn in variables:

Code

```
vec4 currentNormal;
vec3 currentReflectance;
```

which we set in the draw function before drawing a triangle. However, the 3D position that the pixel corresponds to will be different for different pixels of the triangle and we thus have to interpolate it. You therefore need to add it to the Pixel-struct:

Code

```
struct Pixel
{
    int x;
    int y;
    float zinv;
    vec4 pos3d;
};
```

Since we will no longer interpolate the illumination we do not need a variable for it in the Pixel-struct. Also since we will not compute the illumination in the VertexShader you can remove the normal and reflectance from the Vertex-struct, since they will not be used:

Code

```
struct Vertex
{
    vec4 position;
};
```

Instead of computing the illumination in VertexShader you just need to store the 3D position of the Vertex to the corresponding variable in Pixel. Then you should interpolate this value in Interpolation instead of illumination, but in the same way though. You will then have access to the 3D position of the pixel in PixelShader and you can compute the illumination. You should then get the image seen in figure 6.

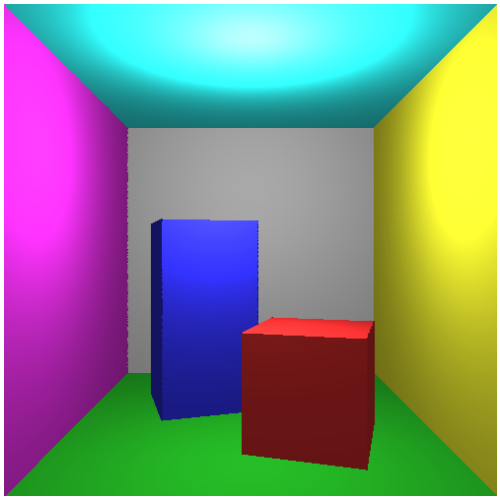

Figure 6: Unique illumination computed for every pixel.

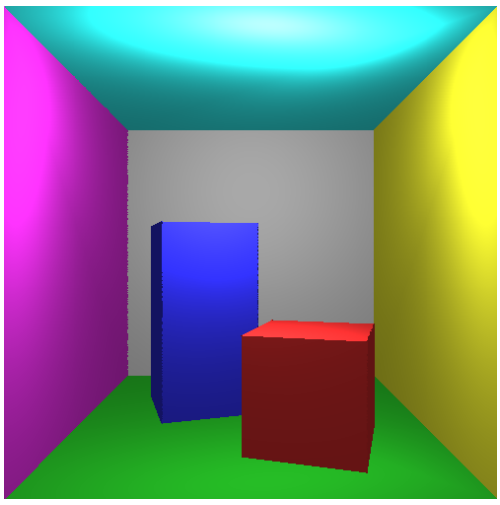

Figure 7: Unique illumination computed for every pixel, but not using perspective correct interpolation for the 3D position of the pixel.

However, it is likely that you will instead get the result seen in figure 7, where the illumination looks a bit skew. This happens if you do not implement perspective correct interpolation, but just interpolate the

3D position linearly. As long as the light source does not move this does not look too bad, but if the light is moving it does not look that great.

When you interpolated the depth z over the image you could not interpolate z linearly. Instead you had to us the quantity $1/z$ to have something that changes linearly in the image. In general if you have a some quantity q that changes linearly in the 3D world, it will not change linearly in an image created by perspective projection. Instead q/z will change linearly in the image. For instance, if we want to interpolate the 3D position p we can interpolate p/z linearly, and then after interpolation we can multiply it with z, or equivalently divide with $1/z$, to get the interpolated 3D position that we want. Make sure that you do this when you interpolate pixels.

Finally, make sure that you can move the light source using the a,w,s,d keys as in the second lab and you are done with this lab as well. Your rasterizer should then render the same image as your final raytracer except for shadows. This is one of the things that are more difficult to compute in a rasterizer.

I am sure that doing this lab felt a lot less straight forward compared to the raytracer we did before. I am also pretty certain that your rendering is a lot more efficient here and that moving the camera interactively actually feels somewhat responsive. The rendering we looked at here are what is used in games and otherrealtime applications. Because speed is essential a lot of rasterization is now done directly in hardware to speed up things even more. Rasterization libraries such as OpenGL and Microsoft's DirectX are able to utilise hardware acceleration and these will be the APIs that you are most likely to use in the future ifyou are interested in programming interactive realtime graphics such as games. The important thing isnow you really understand and know how these things works from the bottom up and I think you will beable to start using such APIs in no time at all and importantly much more efficiently as you know what happens underneath the shiny surface. If you are interested in getting started with OpenGL I reallyrecommend the tutorials on NeHe. I think that you should be able to very quickly pick up how these things work. As a last thing go back to the abstract and read it through once more, do you feel like you have picked up the things that are outlined there, if so, you should be very proud, you have written a big chunk of code and you have a very good grasp on what computer graphics really are. Well done!

7 Extensions

There are lots of things that we can add to our rasterizer to make it look more interesting have a think about how you would add things such as,

- Shadows
- Loading of general models
- Textures
- Normal mapping
- Parallax mapping
- optimizations
- level-of-detail
- hierarchical models
- efficient data-structures
- more advanced clipping
- etc.

Figuring out how to extend a rasteriser is not at all as obvious as it is when it comes to raytracing, a physics book is not going to tell you all you need to know. Actually do not think about how it physically works, think about how "physics" looks and see you can come up with ways of achieving similar effects with more efficient methods. To me this is why rasterisation is such a fun thing to code, it is really about hacking together something. Have a look at your favourite game and think about how different things could be done, can you figure it out and if so maybe you should have a try at implementing it.

Good Luck and Happy Hacking