Computer Graphics SoSe16
Exercise 6

Tobias Knopp & Martin Hofmann
Institute for Biomedical Imaging

Next exercises:
04.07.2016, 14:15-15:45 & 16:30-18:00
11.07.2016, 14:45-15:45 & 16:30-18:00

Now that we have our own rasterization stage available the aim of this exercise sheet will be to put together all the pieces to build a complete graphics pipeline.

You may use all the methods, we used so far (see `CG.jl`).

1. At first we revisit the second exercise sheet, were we rendered the `HouseOfSantaClaus` using PyPlot for the viewport transformation and the line drawing. Your task will be to implement a new render function which uses our custom line drawing algorithm to render the `HouseOfSantaClaus`.

   Starting with the original vertex data of the `houseOfSantaClaus` we have to apply a series of transformations before we are able to start drawing lines.

   (a) First, we apply a model transformation the transform our `HouseOfSantaClaus` to the world space coordinate system. This was already done e.g. in task 6 of sheet 2 by `scaling(0.5,0.5,0.5)*houseOfSantaClaus`.

   (b) Second, we have to apply a camera transformation to transform from world to camera space. This could be done using the `camera.worldToCam` transformation.

   (c) Third, a projection transformation is applied, which projects all vertices into the canonical viewing volume $[-1,1]\times[-1,1]\times[-1,1] \subset \mathbb{R}^3$. Within the camera coordinate system the viewing direction is $-z$ and the viewing volume is given by $[l,r] \times [b,t] \times [f,n]$

   - $u = l$ is the left plane
   - $u = r$ is the right plane
   - $y = b$ is the bottom plane
   - $y = t$ is the top plane
   - $z = n$ is the near plane
   - $z = f$ is the far plane

   To transform this viewing volume into the canonical viewing volume one can use

$$M_{orth} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1}$$

For our `OrthoCamera` we assumed the viewing volume to already coincide with the canonical viewing volume, so M_{orth} is the unit matrix.

(d) Last we have to apply a viewport transformation, which maps the vertex positions from our canonical viewing volume to screen space (near plane $z = -1$). We project $x = -1$ to the left side of the screen, $x = +1$ to the right side of the screen, $y = -1$ to the bottom of the screen, and $y = +1$ to the top of the screen. If we are drawing into a `FrameBuffer` that has `nx` by `ny` pixels(pixel $1, 1$ ate the top left and pixel nx,ny at the bottom right), we need to map the square $[-1, 1]^2$ to the rectangle $[0.5, nx + 0.5] \times [0.5, ny + 0.5]$. This can be done by

$$\begin{pmatrix} x_{buffer} \\ y_{buffer} \\ z_{buffer} \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x+1}{2} \\ 0 & -\frac{n_y}{2} & 0 & \frac{n_y+1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{pmatrix}. \tag{2}$$

In screen space (x_{buffer}, y_{buffer}) can then be rounded to their nearest integer positions and lines can be drawn between them using the Bresenham's line algorithm. far plane is at $f = -\mathtt{d}-2$ in our example.

Write a `renderPipeline!` function, which takes the original vertex data (`Object`), a model transformation `Transformation`, an `Orthocamera`, and a `FrameBuffer` to draw the lines from point to point into the `FrameBuffer`. Before drawing you will have to apply all of the transformations above. Note that the transformation 4 is related to the `FrameBuffer`, whereas the transformations 2 and 3 are related to the `Camera`. Do not perform any clipping Operations.

2. Rerender the `houseOfSantaClaus` as done in exercise 2 task 6.

```
# model transformation scales down the house
modelTransformation = scaling(0.45,0.45,0.45)
# canonical view direction
camera = OrthoCamera(Float32[0,0,1],Float32[0,0,-1],Float32[0,1,0])
# initialize Frame Buffer with 500x500 pixels
buffer = FrameBuffer(500,500)
# render scene into buffer
renderPipeline!(houseOfSantaClaus,modelTransformation,camera,buffer)
plot(buffer)
sleep(1)

# camera moved backwards 9 unit length
clear!(buffer)
camera = OrthoCamera(Float32[0,0,10],Float32[0,0,-1],Float32[0,1,0])
renderPipeline!(houseOfSantaClaus,modelTransformation,camera,buffer)
plot(buffer)
sleep(1)

# rotate screen clockwise
for t=0:60
    clear!(buffer)
    camera = OrthoCamera(Float32[0,0,1],Float32[0,0,-1],Float32[sin(2*pi*t
        /60),cos(2*pi*t/60),0])
```

```
    renderPipeline!(houseOfSantaClaus,modelTransformation,camera,buffer)
    plot(buffer)
    sleep(0.01)
end
```

3. In exercise 3 a `PinholeCamera` was introduced. In this exercise we use the convention that the eye position is located at the origin and the viwing direction is $-z$. The near and far planes that limit the range of distances to be seen. In this context, we will use the near plane as the projection plane, so the image plane is at $n = -\mathtt{d}$ and the far plane is at $f = -\mathtt{d}-2$ in our example.

   To be able to reuse most of the methods above we add the perspective transformation P in front of the projection transformation given by equation (1). P maps the perspective viewing volume into the viewing volume $[-\frac{w}{2}, \frac{w}{2}] \times [-\frac{h}{2}, \frac{h}{2}] \times [n, f]$. It is given by

$$P = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3}$$

   After this projection operation has been performed we are basically handling the case of a `OrthoCamera` and all that is left to do, is to perform the projection transformation given by equation (1) and the viewport transformation given by equation (2).

   The special transformation in equation (3) however also changes the 4th component of our four vectors (x, y, z, w), such that after all the transformations have been applied, vectors have to be normalized to have $w = 1$ before their coordinates x, y and z can be used.

   Take the above mentioned into account to write a `renderPipeline!`, which is able to handle a `PinholeCamera`.

4. Rerender the `houseOfSantaClaus` for differnt camera positions. What happens to the Object as the camera moves further away?

```
# move camera towards the houseOfSantaClaus
for t=0:60
    clear!(buffer)
    camera = PinholeCamera(Float32[0,0,10-9*t/60],Float32[0,0,-1],Float32
        [0,1,0])
    renderPipeline!(houseOfSantaClaus,modelTransformation,camera,buffer)
    plot(buffer)
    sleep(0.01)
end
```

5. For real 3D rendering it is of cause not sufficient to render lines. What we would like to do is rendering triangles using the `drawTriangle!` method from the last exercise.

   For that we need a convention to process the vertex data into triangles. In our case we choose to take blocks of 3 successive vertices stored in our `Object` type as vertices to draw triangles. Using this convention we can draw the `houseOfSantaClaus` using the following 9 vertices (corresponding 3 triangles):

```
v1 = Vec4f(-1,-1,0,1)
v2 = Vec4f(1,-1,0,1)
v3 = Vec4f(1,1,0,1)
v4 = Vec4f(1,1,0,1)
v5 = Vec4f(-1,1,0,1)
v6 = Vec4f(-1,-1,0,1)
v7 = Vec4f(-1,1,0,1)
v8 = Vec4f(0,2,0,1)
v9 = Vec4f(1,1,0,1)
houseOfSantaClaus = Object(v1,v2,v3,v4,v5,v6,v7,v8,v9)
```

Write a method similar to the `renderPipeline!` and render the `houseOfSantaClaus` by drawing triangles from the vertex data into the `frameBuffer` and render the scene:

```
clear!(buffer)
# model transformation scales down the house
modelTransformation = scaling(0.45,0.45,0.45)
# canonical view direction
camera = OrthoCamera(Float32[0,0,1],Float32[0,0,-1],Float32[0,1,0])
# render the triangles into buffer
renderTrianglePipeline!(houseOfSantaClaus,modelTransformation,camera,buffer)
plot(buffer)
```

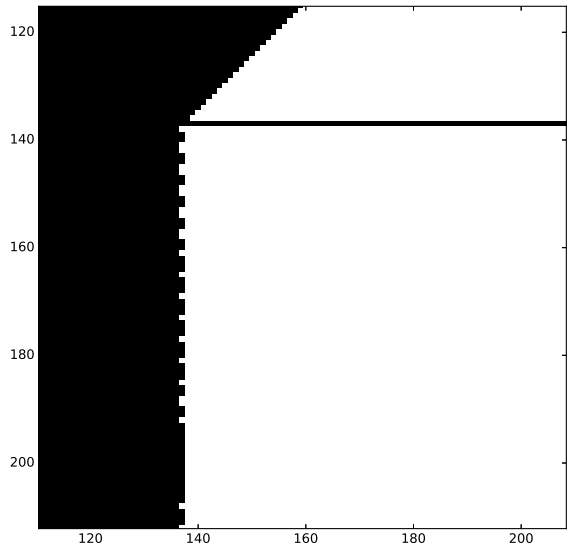

Figure 1: The resulting image will have have some visual artifacts at the edges of the triangles. Why do they occur? How to remove them?