Computer Graphics
Hand-in-5

xFrednet

November 10, 2020

1 TGRA 2017

1.1 10% Shaders and lighting (2017 No. 02)

Describe the difference between per-vertex lighting and per-fragment/per-pixel lighting. In each case bear the following questions in mind:

- What are the advantages and drawbacks of the method?
- What calculations happen where,and what values are set to the final result?
- How is the data processed between different parts of the calculations?

Per-vertex Lighting	**Per-fragment lighting**
The entire lighting calculations are done for each vertex. The resulting light or color is than passed to the fragment shader. The fragment shader input gets interpolated by OpenGL depending on the actual pixel position.	Some world position dependent values are calculated in the fragment shader. The actual lighting calculation is done in the fragment shader is self for each pixel. The input are the interpolated outputs of the fragment shader
The only data that is being passed from the vertex shader to the pixel shader is the object color or lighting effect if the color gets applied in the fragment shader	The world position and normal vector has to be passed to the fragment shader to calculate the lighting effect. The fragment shader usually already calculates the vector v (vector to camera) and S (vector to light) to reduce some calculations for the fragment shader. These values are all passed to the fragment shader.
+ Fewer calculations since lighting is only calculated for each vertex - A bit inaccurate as the light is only calculated for a few spots and than interpolated between + difference can be unnoticeable on objects with a lot of vertices and small faces. - This is done before clipping. (Can cost more performance depending on the setup) + Sends less data through the pipeline	- More calculations because lighting is calculated fpr each fragment / pixes + As accurate as because each pixel is calculated individually + Works on faces with any size + Done after clipping only for pixels that are visible. - Sends more data through the pipeline

1.2 (10%)Cohen-Sutherland Clipping (2017 No. 3)

A clipping window has the following geometry:

- Window(left, right, bottom, top) = (200, 600, 100, 400)

A line with the following end points is drawn in the world:

P1: (650, 450)

P2: (350, 300)

Show how the Cohen-Sutherland clipping algorithm will clip these lines and what their final endpoints, if any, are. Show the coordinate values of P1 and P2 after each pass of the algorithm.

---

v \ >	**left**	**center**	**right**
top	1010	0010	0110
center	1000	0000	0100
bottom	1001	0001	0101

```
Iteration 0:
    \\ P1: (650, 450)
    \\ P2: (350, 300)

Iteration 1:

    Bitmaks
        P1 = 0110
        P2 = 0000

    P1 | P2 != 0000 -> Something is out of bounds
    P1 & P2 == 0 -> We need to clip

    Clipping P1 from top:
        P1.x += (top - P1.y) * (P2.x - P1.x) / (P2.y - P1.y) // -100
        P1.y = 400

    \\ P1 = (550, 400)
    \\ P2 = (350, 300)

Iteration 2:

    Bitmasks:
        P1 = 0000
        P2 = 0000

    P1 | P2 == 0 -> Both points are inside the window
    \\ -> The line gets drawn the final positions are:
    \\      P1 = (550, 400)
    \\      P2 = (350, 300)
```

1.3 (25%) Matrices and transformations (2017 No 5)

1.3.1 (10%) Set up the values in a matrix that represents this position and orientation of a camera. Which matrix in your shader should be set to these values?

A camera is set up to be positioned in (5,3,7) looking at the point (0,0,0). It has an up vector (0,1,0).

$$eye = \begin{pmatrix} 5 \\ 3 \\ 7 \end{pmatrix}$$

$$n = normalize(eye - look\_at) = normalize \begin{pmatrix} 5-0 \\ 3-0 \\ 7-0 \end{pmatrix} = \begin{pmatrix} 0.5488 \\ 0.3293 \\ 0.7684 \end{pmatrix}$$

$$u = normalize\,(up \times n) = \begin{pmatrix} 0.8137 \\ 0 \\ -0.5812 \end{pmatrix}$$

$$v = normalize\,(n \times u) = \begin{pmatrix} -0.1914 \\ 0.9442 \\ -0.2680 \end{pmatrix}$$

$$view\_matrix = \begin{pmatrix} 0.8137 & 0 & -0.5812 & 0.0 \\ -0.1914 & 0.9442 & -0.2680 & 0.0 \\ 0.5488 & 0.3293 & 0.7684 & -9.1104 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The view matrix should be set to these values.

1.3.2 (5%) Find the exact values for a matrix that calculates this camera. Which matrix in your shader should be set to these values?

The camera should have a field of view of 90 degrees, an aspect ratio of 16:9, a near plane at 10 and a far plane at 110.

WRONG TAN USE tan(angle / 2 * PI / 180)

N = 10

F = 110

angle = 90 * PI / 180

aspect = $\frac{16}{9}$

top = $N * tan\left(\frac{angle}{2}\right) = 10 * 1.0 = 10.0$

bottom = -top = -10

right = top * aspect = 17.7778

left = -right = -17.7778

$$\begin{pmatrix} 0.5625 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & -1.2 & -22 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

The projection matrix would be set to these values.

1.3.3 (5%) Represent this coordinate frame in a matrix. Which matrix would this commonly be?

Vertex data should be drawn into a coordinate frame that has been translated by (1, 7, 3) and then rotated by 150 degrees about the y-axis

---

$$\begin{pmatrix} -0.8660 & 0 & 0.5 & 1 \\ 0 & 1 & 0 & 7 \\ -0.5 & 0 & -0.8660 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The model matrix would usually be set to these values. (Also known as the transformation matrix)

1.3.4 (5%) Given the matrix values calculated in parts a, b & c, what values will the vertex shader set to gl_Position?
Will this vertex be within the viewing volume and thus be rendered as part of the final image? Explain.

vertex position = (2, 3, 1)

$$world\_position = \begin{pmatrix} -0.8660 & 0 & 0.5 & 1 \\ 0 & 1 & 0 & 7 \\ -0.5 & 0 & -0.8660 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 2 \\ 3 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.8660 * 2 + 0.5 * 1 + 1 * 1 \\ 1 * 3 + 7 * 1 \\ -0.5 * 2 + -0.8660 * 1 + 3 * 1 \\ 1 * 1 \end{pmatrix} = \begin{pmatrix} -0.2320 \\ 10 \\ 1.134 \\ 1 \end{pmatrix}$$

$$eye\_position = \begin{pmatrix} 0.8137 & 0 & -0.5812 & 0.0 \\ -0.1914 & 0.9442 & -0.2680 & 0.0 \\ 0.5488 & 0.3293 & 0.7684 & -9.1104 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} -0.2320 \\ 10 \\ 1.134 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.8479 \\ 9.1824 \\ -5.0734 \\ 1 \end{pmatrix}$$

$$GL\_Position = \begin{pmatrix} 0.56 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & -1.2 & -22 \\ 0 & 0 & -1 & 0 \end{pmatrix} * \begin{pmatrix} -0.8479 \\ 9.1824 \\ -5.0734 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.4748 \\ 9.1824 \\ 15.9119 \\ 5.0734 \end{pmatrix} = \begin{pmatrix} -0.4748/5.0734 \\ 9.1824/5.0734 \\ 15.9119/5.0734 \\ 5.0734/5.0734 \end{pmatrix} = \begin{pmatrix} -0.0936 \\ 1.8099 \\ 2.7750 \\ 1 \end{pmatrix}$$

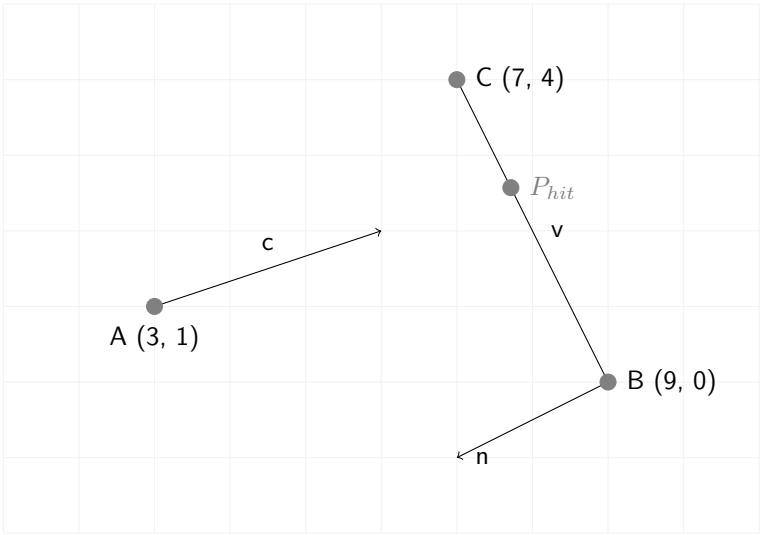
The vertex would be clipped because both the y and the z value are > 1 and therefore outside of the screen.

1.4 (10%) Vector intersections and reflections

A line has end points (9, 0) and (7, 4).

A particle starts at (3, 1) and travels along in the direction (6, 2).

1.4.1 (7%) In which point does the path of the particle cross the line?



$v = C - B = \begin{pmatrix} -2 \\ 4 \end{pmatrix}$

Line: $B = \begin{pmatrix} 9 \\ 0 \end{pmatrix}$ $n = v \perp = \begin{pmatrix} -v.y \\ v.x \end{pmatrix} = \begin{pmatrix} -4 \\ -2 \end{pmatrix}$

Particle: $A = \begin{pmatrix} 9 \\ 0 \end{pmatrix}$ $c = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$

$$t_{hit} = \frac{n*(B-A)}{n*c} = \frac{\begin{pmatrix} -4 \\ -2 \end{pmatrix} \circ \begin{pmatrix} 9-3 \\ 0-1 \end{pmatrix}}{\begin{pmatrix} -4 \\ -2 \end{pmatrix} \circ \begin{pmatrix} 6 \\ 2 \end{pmatrix}} = \frac{-4*6+-2*-1}{-4*6+-2*2} = \frac{-22}{-28} = \frac{11}{14}$$

$$P_{hit} = A + t_{hit} * c = \begin{pmatrix} 3 \\ 1 \end{pmatrix} + \frac{11}{14} * \begin{pmatrix} 6 \\ 2 \end{pmatrix} = \begin{pmatrix} 7.7143 \\ 2.5714 \end{pmatrix}$$

1.4.2 (3%) If the particle is made to bounce off the line, what will its new direction vector be?

$$r = c - 2 * \frac{(con)}{non} * n = \begin{pmatrix} 6 \\ 2 \end{pmatrix} - 2 * \frac{-28}{20} * \begin{pmatrix} -4 \\ -2 \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \end{pmatrix} - \begin{pmatrix} 11.2 \\ 5.6 \end{pmatrix} = \begin{pmatrix} -5.2 \\ -3.6 \end{pmatrix}$$

1.5 (10%) Bezier motion

Scalars in bezier curves can befound by factoring Bernstein polynomials: BL $= ((1-t)+t)^L$ for a bezier curve with L + 1 control points.

The camerais moved along a bezier curve with 4 control points.

P1: (7, 3, 2)

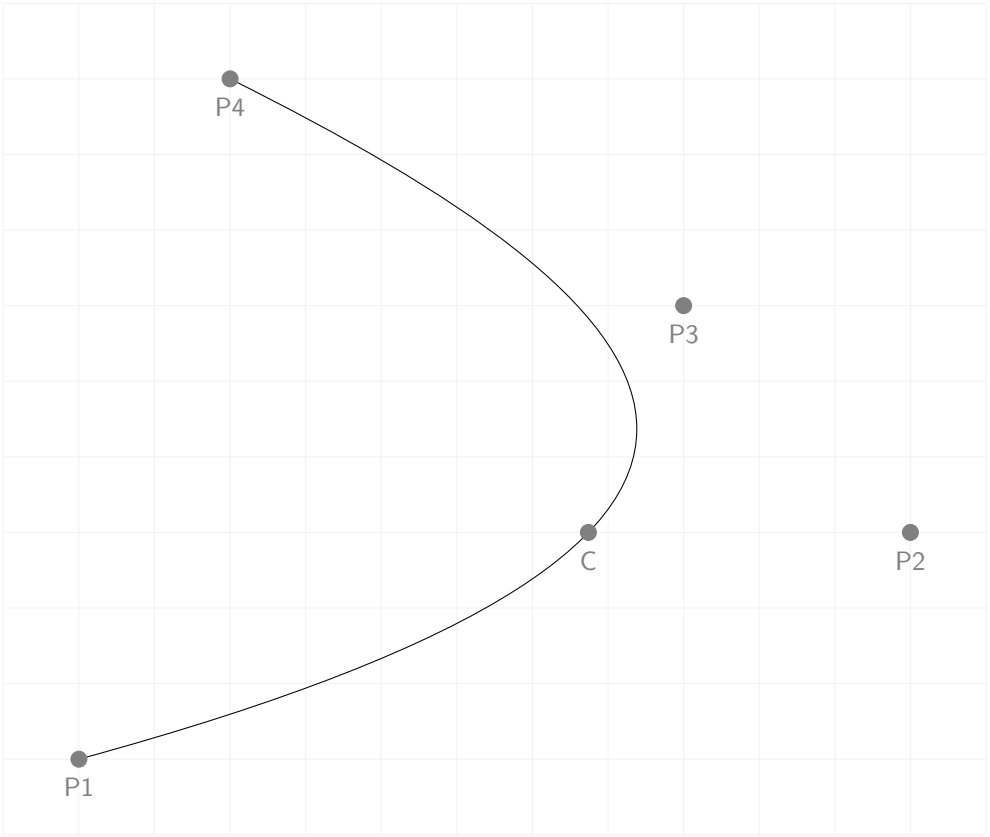
P2: (18, 3, 5)

P3: (15, 3, 8)

P4: (9, 3, 11)

The motion should start 14 seconds after the program starts and it should end 24 seconds later, 38 seconds after the program starts.

1.5.1 What is the camera's position 22 seconds after the program started?



$t = \frac{t_{now} - t_{start}}{t_{duration}} = \frac{22-14}{24} = \frac{1}{3}$

$P = (1-t)^3 * t^0 * P1 + 3 * (1-t)^2 * t^1 * P2 + 3 * (1-t)^1 * t^2 * P3 + (1-t)^0 * t^3 * P4$

$$P = \frac{8}{27} * \begin{pmatrix} 7 \\ 3 \\ 2 \end{pmatrix} + \frac{4}{9} * \begin{pmatrix} 18 \\ 3 \\ 5 \end{pmatrix} + \frac{2}{9} * \begin{pmatrix} 15 \\ 3 \\ 8 \end{pmatrix} + \frac{1}{27} * \begin{pmatrix} 9 \\ 3 \\ 11 \end{pmatrix}$$

$$P = \begin{pmatrix} 13.74 \\ 3 \\ 5 \end{pmatrix}$$

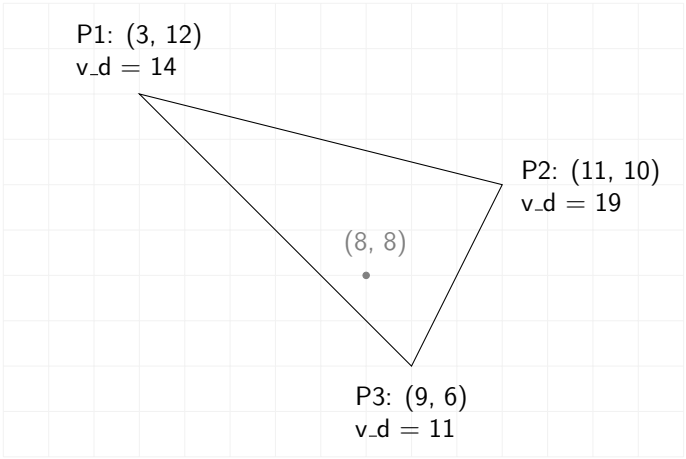
2 TGRA Hand-in 5

2.1 6. (10%) Rasterization (2015, No. 6)

Three vertices of a triangle have been sent through the OpenGL pipeline. They have the following pixel positions as well as values for the varying variable v_d:

- P1: position = (3, 12) – v_d = 14
- P2: position = (11, 10) – v_d = 19
- P3: position = (9, 6) – v_d = 11

2.1.1 What will the fragment shader value of v_d be set to at pixel (8,8)?



$y_{bot} = 6$

$y_{top} = 12$

$x_{left} = lerp\left(P3.x, P1.x, \frac{y-P3.y}{P1.y-P3.y}\right) = lerp\left(9, 3, \frac{8-6}{12-6}\right) = lerp\left(9, 3, \frac{1}{3}\right) = 9 * \left(1 - \frac{1}{3}\right) + 3 * \frac{1}{3} = 6 + 1 = 7$

$x_{right} = lerp\left(P3.x, P2.x, \frac{y-P3.y}{P2.y-P3.y}\right) = lerp\left(9, 11, \frac{8-6}{10-6}\right) = lerp\left(9, 11, \frac{1}{2}\right) = 9 * \left(1 - \frac{1}{2}\right) + 11 * \frac{1}{2} = 4.5 + 5.5 = 10$

$v\_d_{left} = lerp\left(P3.v\_d, P1.v\_d, \frac{y-P3.y}{P1.y-P3.y}\right) = lerp(11, 14, \frac{1}{3}) = 11 * \left(1 - \frac{1}{3}\right) + 14 * \frac{1}{3} = 12$

$v\_d_{right} = lerp\left(P3.v\_d, P2.v\_d, \frac{y-P3.y}{P2.y-P3.y}\right) = lerp(11, 19, \frac{1}{2}) = 11 * \left(1 - \frac{1}{2}\right) + 19 * \frac{1}{2} = 15$

$v\_d_{pixel} = lerp\left(v\_d_{left}, v\_d_{right}, \frac{x_{pixel}-x_{left}}{x_{right}-x_{left}}\right) = lerp\left(12, 15, \frac{1}{3}\right) = 12 * \left(1 - \frac{1}{3}\right) + 15 * \frac{1}{3} = 13$

Pixel (8, 8) has the v_d value 13. Awesome!

A small story about the beautiful triangle image in this exercise:

Once upon a time, I was procrastinating doing this assignment. I spent some time learning how to work with geometric shaders in OpenGL and developing particle rendering concept. You know the usual weekend routine. On Monday I decided that I had to complete this to hand it in. Well... That image is the result of me procrastinating another hour and learning how to draw in LaTeX.

I hope you have a good day sir!

2.2 (10%) Bezier motion (2012 No. 6)

Scalars in bezier curves are found by factoring Bernstein polynomials:

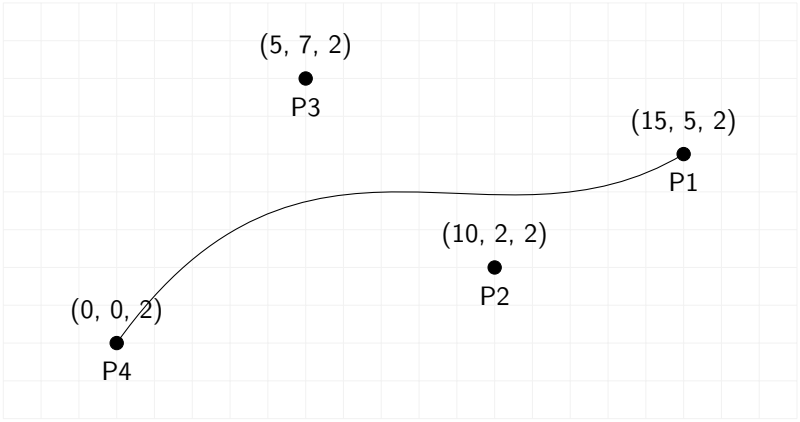
- BL = ((1 - t) + t) L for a bezier curve with L + 1 control points.

An object is moved along a bezier curve with 4 control points.

- P1 = (15, 5, 2)
- P2 = (10, 2, 2)
- P3 = (5, 7, 2)
- P4 = (0, 0, 2)

The motion should start 4 seconds after the program starts and it should end 20 seconds later, 24 seconds after the program starts.

2.2.1 Where is the object's center 19 seconds after the program started?



$t_{start} = 4$

$t_{duration} = 20$

$t_{now} = 19$

$delta = \frac{t_{now} - t_{start}}{t_{duration}} = \frac{15}{20} = \frac{3}{4}$

Iteration 1:

$$\vec{P1}_1 = lerp\,(P1_0, P2_0, delta) = lerp\left(\begin{pmatrix}15\\5\\2\end{pmatrix}, \begin{pmatrix}10\\2\\2\end{pmatrix}, \tfrac{3}{4}\right) = \begin{pmatrix}11.25\\2.75\\2\end{pmatrix}$$

$$\vec{P2}_1 = lerp\,(P2_0, P3_0, delta) = lerp\left(\begin{pmatrix}10\\2\\2\end{pmatrix}, \begin{pmatrix}5\\7\\2\end{pmatrix}, \tfrac{3}{4}\right) = \begin{pmatrix}6.25\\5.75\\2\end{pmatrix}$$

$$\vec{P3}_1 = lerp\,(P3_0, P4_0, delta) = lerp\left(\begin{pmatrix}5\\7\\2\end{pmatrix}, \begin{pmatrix}0\\0\\2\end{pmatrix}, \tfrac{3}{4}\right) = \begin{pmatrix}1.25\\1.75\\2\end{pmatrix}$$

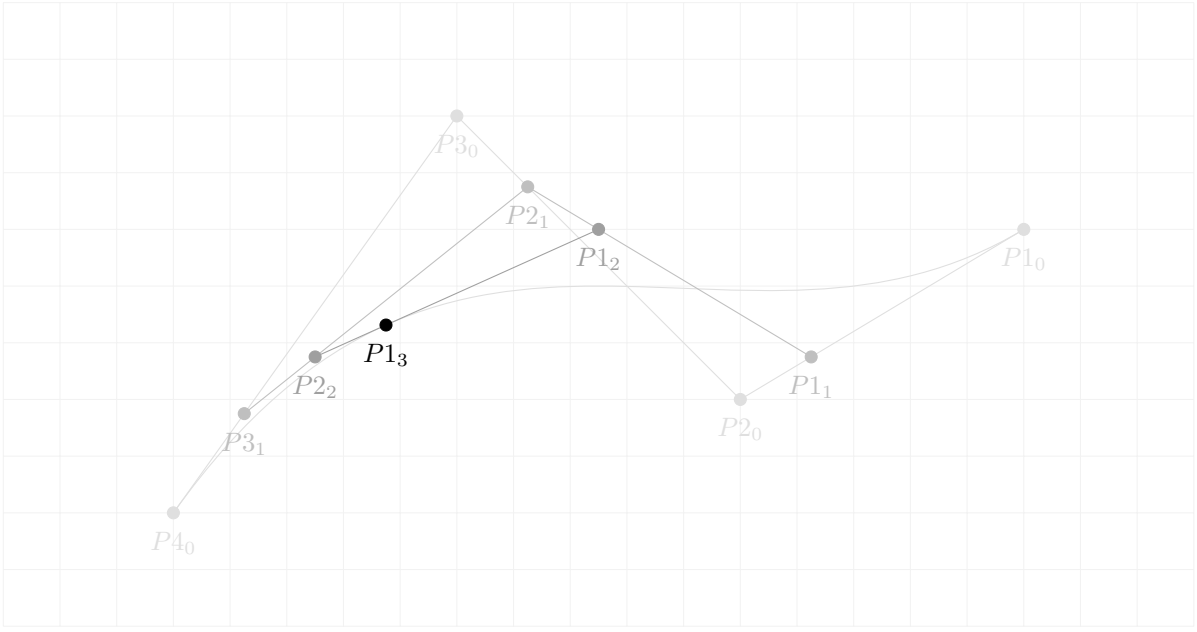
Iteration 2:

$$\vec{P1}_2 = lerp\,(P1_1, P2_1, delta) = lerp\left(\begin{pmatrix}11.25\\2.75\\2\end{pmatrix}\begin{pmatrix}6.25\\5.75\\2\end{pmatrix}\tfrac{3}{4}\right) = \begin{pmatrix}7.5\\5\\2\end{pmatrix}$$

$$\vec{P1}_2 = lerp\,(P1_1, P2_1, delta) = lerp\left(\begin{pmatrix}6.25\\5.75\\2\end{pmatrix}\begin{pmatrix}1.25\\1.75\\2\end{pmatrix}\tfrac{3}{4}\right) = \begin{pmatrix}2.5\\2.75\\2\end{pmatrix}$$

Iteration 3:

$$\vec{P1}_3 = lerp\,(P1_2, P2_2, delta) = lerp\left(\begin{pmatrix}7.5\\5\\2\end{pmatrix}\begin{pmatrix}2.5\\2.75\\2\end{pmatrix}\tfrac{3}{4}\right) = \begin{pmatrix}3.75\\3.3125\\2\end{pmatrix}$$



The object's center is at $\begin{pmatrix}3.75\\3.3125\\2\end{pmatrix}$ when $t = 19$. The calculation took 3 iterations.

This was actually fun to calculate and draw. I have to say that these hand-ins have had a really good learning effect for me. (Let's hope that the results are also correct.) I'm almost sad that this is the last one.

But all good things must end at one point. Thank you very much!

3 TGRA Hand-in 4

3.1 (20%) Shaders (2015 No. 4)

On the following two pages there are shaders for use in an OpenGL pipeline, a vertex shader and a fragment shader. You can add both uniform and varying variables to these shaders above their main() function definition and you can add any number of variables and code inside the main functions.

The following types can be used:

`float`	a floating point number
`vec4`	a 4 coordinate vector, used for positions, directions and colors
`mat4`	a 4x4 matrix

The following GLSL functions can be used:

`float`	`dot(vec4 v1, vec4 v2):`	Returns the dot product of two vectors
`vec4`	`normalize (vec4 v):`	Returns the normalized vector
`float`	`length(vec4 v):`	Returns the length of the vector
`vec4`	`cross(vec4 v1, vec4 v2):`	Returns the cross product of two vectors
`float`	`max(float a, float b):`	Returns the higher value
`float`	`min(float a, float b):`	Returns the lower value
`float`	`pow(float num, float exp):`	Returns num to the power of exp

`+, -, *, /` are done component-wise on vec4 but as matrix multiplications when one or both sides are mat4.

Variables are already defined for the attributes `a_position` and `a_normal` for each vertex that will be sent through the pipeline.

The current shaders both have the same initial definitions at this point. Strike out those that are unnecessary in each shader and add whatever definitions you need for each of the following problems.

3.1.1 a) (5%) Add variables and calculations needed to transform the position attribute (vertices) to global coordinates, eye coordinates and clip coordinates and set the value for the built-in output variable gl_Position.

3.1.2 b) (10%) Add variables and calculations needed for the shaders to do per-fragment lighting. Include ambient, diffuse and specular lighting for a single light source. Set the final color value to the built-in output variable gl_FragColor.

3.1.3 c) (5%) Fog is a color that is mixed with the final fragment color based on distance from the camera. Anything closer than u_startFog is 100% the fragment color, anything further away than u_endFog is 100% the fog color. Anything in between is a weighted average of the two based on the proportion of the distance in the startFog to endFog range. Add variables and calculations to the shaders to implement this functionality.

Vertex shader

```
    // xxx attribute vec3 a_position;
    // xxx attribute vec3 a_normal;

+   in vec3 in_position;
+   in vec3 in_normal;

+   out vec4 surface_normal; // Also called n
+   out vec4 to_light;   // Also called S
+   out vec4 to_camera;  // Also called v

    uniform vec4 u_lightPosition;
+   uniform vec4 u_camera_position;
    // xxx uniform vec4 u_lightColor;

    // xxx uniform vec4 u_materialAmbient;
    // xxx uniform vec4 u_materialDiffuse;
    // xxx uniform vec4 u_materialSpecular;
    // xxx uniform float u_materialShininess;

+   uniform mat4 u_transformation_matrix;
+   uniform mat4 u_view_matrix;
+   uniform mat4 u_projection_matrix;

    main() {
+       vec4 global_coordinates = u_transformation_matrix * vec4(in_position, 1.0);
+       vec4 eye_coordinates = u_view_matrix * eye_coordinates;
+       vec4 clip_coordinates = u_projection_matrix * eye_coordinates;

+       to_light = u_lightPosition - global_coordinates;
+       to_camera = u_camera_position - global_coordinates;

+       surface_normal = normalize(u_transformation_matrix * vec4(in_normal, 0.0));
+       gl_Position = clip_coordinates;
    }
```

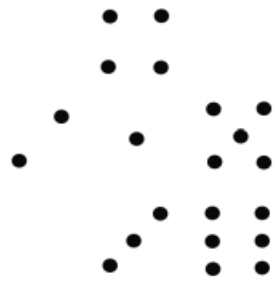
Fragment shader

```
    // xxx attribute vec3 a_position;
    // xxx attribute vec3 a_normal;

+   in vec4 surface_normal;
+   in vec4 to_light;
+   in vec4 to_camera;

    // xxx uniform vec4 u_lightPosition;
    uniform vec4 u_lightColor;

    uniform vec4 u_materialAmbient;
    uniform vec4 u_materialDiffuse;
    uniform vec4 u_materialSpecular;
    uniform float u_materialShininess;

+   uniform vec4 u_startFog;
+   uniform vec4 u_endFog;
+   uniform vec4 u_fog_color;

    main() {
+       // ##############
+       // Lighting
+       // ##############
+       // diffuse lighting
+       float lambert = max(dot(surface_normal * normalize(to_light)), 0.0);
        vec4 diffuse = u_materialDiffuse * lambert;
+
+       // specular lighting
+       vec4 h = to_light + to_camera;
+       float phong = max(dot(surface_normal * normalize(h)), 0.0);
+       vec4 specular = u_materialSpecular * pow(phong, u_materialShininess);

+       // ambient
+       vec4 ambient = u_materialAmbient * u_lightColor;
+       vec4 fragment_color = diffuse + specular + ambient;

+       // #############
+       // Fog
+       // #############
+       float camera_distance = length(to_camera);
+       float fog_diff = u_endFog - u_startFog;
+       float in_fog_distance = camera_distance - u_startFog;
+       float fog_value = max(min(in_fog_distance / fog_diff, 0.0), 1.0);
+
+       // #############
+       // Output
+       // #############
+       //             <--         Light              -->  <--       Fog       -->
+       gl_FragColor = fragment_color * (1.0 - fog_value) + u_fog_color * fog_value;
+       // Looking back on this let's me see some optimizations but I'll keep it like this :)
    }
```

This doesn't look that good with latex tbh... but I'm to stubborn to change it at this point. Stubbornness is not always a good trait.

3.2 Graphics programming (10%) (2013 No. 7)

The class D6 (previous/opposite page) contains data to draw a 3D cube. Imagine that the texture sent into it's constructor is this image:



3.2.1 a) (5%) Add to the class D6 the code needed to use this texture and to map a different numbered face to each of the cube's sides. You can add code between any lines in the code on the opposite page.

Hint: You need to set up the UV coordinates (between 0 and 1) for all the cube's vertices.

3.2.2 b) (5%) A separate dice rolling engine returns a list of the structure RolledD6:

```
public class coords {
    public float x;
    public float y;
    public float z;
}
public class RolledD6 {
    public float size;
    public coords position;
    public coords rotationAxis;
    public float rotationAngle;
}
```

Finish code that will display each of the rolled dice of the size indicated, at the correct position and rotated by the rotation angle around the vector called rotation axis.

```
void drawDiceSet {
    List<RolledD6> diceList = DiceRollingEngine.getDice();
    foreach(RolledDice d in diceList) {
+       glTranslate(d.position.x, d.position.y, d.position.z);
+       glRotate(d.rotationAngle, d.rotationAxis.x, d.rotationAxis.y, d.rotationAxis.z);
+       glScale(d.size, d.size, d.size);
    }
}
```

```
public class D6 {
    FloatBuffer vertexBuffer;
    FloatBuffer texCoordBuffer;
    Texture texture;

    public D6(Texture tex)
    {
        vertexBuffer = BufferUtils.newFloatBuffer(72);
        vertexBuffer.put(new float[] {
            -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, -0.5f, // back  | 6
             0.5f, -0.5f, -0.5f,  0.5f,  0.5f, -0.5f, //
             0.5f, -0.5f, -0.5f,  0.5f,  0.5f, -0.5f, // left  | 2
             0.5f, -0.5f,  0.5f,  0.5f,  0.5f,  0.5f, //
             0.5f, -0.5f,  0.5f,  0.5f,  0.5f,  0.5f, // front | 1
            -0.5f, -0.5f,  0.5f, -0.5f,  0.5f,  0.5f, //
            -0.5f, -0.5f,  0.5f, -0.5f,  0.5f,  0.5f, // right | 5
            -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, -0.5f, //
            -0.5f,  0.5f, -0.5f, -0.5f,  0.5f,  0.5f, // up    | 4
             0.5f,  0.5f, -0.5f,  0.5f,  0.5f,  0.5f, //
            -0.5f, -0.5f, -0.5f, -0.5f, -0.5f,  0.5f, // down  | 3
             0.5f, -0.5f, -0.5f,  0.5f, -0.5f,  0.5f  //
        });
        vertexBuffer.rewind();

        texCoordBuffer = BufferUtils.newFloatBuffer(48);
        texCoordBuffer.put(new float[] {
+           0.66f, 0.66f,    1.00f, 0.66f, // back  | 6
+           0.66f, 1.00f,    1.00f, 1.00f, //
+           0.00f, 0.33f,    0.33f, 0.33f, // left  | 2
+           0.00f, 0.66f,    0.33f, 0.66f, //
+           0.33f, 0.33f,    0.66f, 0.33f, // front | 1
+           0.33f, 0.66f,    0.66f, 0.66f, //
+           0.66f, 0.33f,    1.00f, 0.33f, // right | 5
+           0.66f, 0.66f,    1.00f, 0.66f, //
+           0.33f, 0.00f,    0.66f, 0.00f, // up    | 4
+           0.33f, 0.33f,    0.66f, 0.33f, //
+           0.33f, 0.66f,    0.66f, 0.66f, // down  | 3
+           0.33f, 1.00f,    0.66f, 0.66f //
        });
        texCoordBuffer.rewind();
+       this.texture = tex;
    }
    public void draw()
    {
        Gdx.gl11.glVertexPointer(3, GL11.GL_FLOAT, 0, vertexBuffer);
+       Gdx.gl11.glTexCoordPointer(2, GL11.GL_FLOAT, 0, texCoordBuffer);
+       Gdx.gl11.glActiveTexture(GL11.GL_TEXTURE0);
+       Gdx.gl11.glBindTexture(GL11.GL_TEXTURE_2D, texture);

        Gdx.gl11.glNormal3f(0.0f, 0.0f, -1.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 0, 4);
        Gdx.gl11.glNormal3f(1.0f, 0.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 4, 4);
        Gdx.gl11.glNormal3f(0.0f, 0.0f, 1.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 8, 4);
        Gdx.gl11.glNormal3f(-1.0f, 0.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 12, 4);
        Gdx.gl11.glNormal3f(0.0f, 1.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 16, 4);
        Gdx.gl11.glNormal3f(0.0f, -1.0f, 0.0f);
        Gdx.gl11.glDrawArrays(GL11.GL_TRIANGLE_STRIP, 20, 4);

+       // I don't unbind the things here because there isn't much space in the
+       // exam and it'll just be overwritten by the next render :)
    }
}
```

I'm not sure if these are the additions that where expected for this task. However, I believe that this should render the dice correctly :)

4 TGRA Hand-in 3

4.1 Lighting calculations (10%) (2014 No. 6)

A single light is in the light model in an OpenGL program. It has the ambient values (0.0, 0.0, 0.0), diffuse values (0.5, 0.3, 0.7), specular values (0.3, 0.8, 0.7) and position (5.0, 8.0, -1.0). There is also a global ambient factor of (0.3, 0.2, 0.4) in the light model. A camera is positioned in (4.0, 6.0, 5.0) and looks towards P.

P has the color values: ambient (0.4, 0.2, 0.3), diffuse (0.4, 0.7, 0.2) and specular (0.6, 0.6, 0.6). It has a shininess value of 13. It has the position (4.0, 4.0, 3.0) and a normal (0.0, 1.0, 0.0).

4.1.1 What will be the blue color value for P on the screen ?

I'm going to calculate the RGB value for the given point. This is not required but probably a good practice :)

$$n = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$S = light_{pos} - P = \begin{pmatrix} 1 \\ 4 \\ -4 \end{pmatrix}$$

diffused lighting

$$lambert = max\left(\frac{n \circ S}{|n| * |S|}, 0\right) = max\left(\frac{4}{\sqrt{33}}, 0\right) \approx 0.6963$$

specular lighting

$$v = camera_{pos} - P = \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}$$

$$h = v + S = \begin{pmatrix} 1 \\ 6 \\ -2 \end{pmatrix}$$

$$phong = max\left(\frac{n \circ h}{|n| * |h|}\right) = max\left(\frac{6}{\sqrt{41}}, 0\right) \approx 0.9370$$

$$f = 13(shininess)$$

light

$$I = (I_d * material_d * lambert) + \left(I_s * material_s * phong^f\right) + (I_a * material_a) + (I_{gaf} * material_a)$$

$$I \approx \left(\begin{pmatrix} 0.5 \\ 0.3 \\ 0.7 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.7 \\ 0.2 \end{pmatrix} * 0.6963\right) + \left(\begin{pmatrix} 0.3 \\ 0.8 \\ 0.7 \end{pmatrix} * \begin{pmatrix} 0.6 \\ 0.6 \\ 0.6 \end{pmatrix} * 0.9370^{13}\right) + \left(\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.2 \\ 0.3 \end{pmatrix}\right) + \left(\begin{pmatrix} 0.3 \\ 0.2 \\ 0.4 \end{pmatrix} * \begin{pmatrix} 0.4 \\ 0.2 \\ 0.3 \end{pmatrix}\right)$$

$$I \approx \left(\begin{pmatrix} 0.20 \\ 0.21 \\ 0.14 \end{pmatrix} * 0.6963\right) + \left(\begin{pmatrix} 0.18 \\ 0.48 \\ 0.42 \end{pmatrix} * 0.9370^{13}\right) + \begin{pmatrix} 0.12 \\ 0.04 \\ 0.12 \end{pmatrix}$$

$$I \approx \begin{pmatrix} 0.3365 \\ 0.3922 \\ 0.3977 \end{pmatrix}$$

The blue value of the object in point P is $\approx$ 0.3977

4.2 (10%) Cohen-Sutherland Clipping (2015 No. 2)

A clipping window has the following geometry:

- Window(left, right, bottom, top) = (200, 600, 100, 400)

A line with the following end points is drawn in the world:

- P1: (240, 480)
- P2: (140, 300)

4.2.1 Show how the Cohen-Sutherland clipping algorithm will clip these lines and what their final endpoints, if any, are. Show the coordinate values of P1 and P2 after each pass of the algorithm.

	left	**center**	**right**
top	1001	1000	1010
center	0001	0000	0010
bottom	0101	0100	0110

Iteration 1

```
P1_code = 1000
P2_code = 0001

// => Clip P1 to the top

P1.x += (top - P1.y) * (P2_x - P1_x) / (P2_y - P1_y) // -44.4444
P1.y = top

// P1: (195.5, 400)
// P2: (140  , 300)
```

Iteration 2

```
P1_code = 0001
P2_code = 0001

// Both points are outside on the same side
// => The entire line gets clipped. The points from the last
//    Iteration still stand
//      * P1: (195.5, 400)
//      * P2: (140  , 300)
```

Okay, this is the end of this hand-in. I have to say that I've gotten used to writing LaTeX by now. It's still slower than writing by hand or in the case of No.2 just using a monospace environment. But this is a good exercise and I had some fun writing these 300 lines ^^

5 TGRA Hand-in 2

5.1 (10%) Transformations and matrices (2015 No. 1)

You have access to the same matrix and graphics classes as we built together this semester, that is a BoxGraphic class with a drawSolidCube that sends vertices into the pipeline for a cube of the size 1x1x1, centered in (0,0,0), and a ModelMatrix class with the basic transformation operations, matrix stack operations and shader manipulation operations. These classes have been created and initialized elsewhere. You don't need to remember the exact names of functions, just the idea behind each one you need to call, and what parameters they take.

What you want to draw is the following scene:

a) A box of size 2x2x2, centered in (9,5,-2) and under it a big flat floor which is 10x10 in the x-z plane but only 0.8 thick. Its corner should be in (0,0,0) and its top edge level with the base x-z plane.

5.1.1 (5%) Write the code that would draw the scene described. Imagine you're already inside the display function and lights, colors and cameras have already been set.

I assume that the BoxGraphic instances have been setup with the following scale and position class members: I ignore the rotation because it wasn't mentioned :)

$$Scale_{Box} = \begin{pmatrix} 2.0 \\ 2.0 \\ 2.0 \end{pmatrix} Position_{Box} = \begin{pmatrix} 9.0 \\ 5.0 \\ -2.0 \end{pmatrix}$$

$$Scale_{Plane} = \begin{pmatrix} 10.0 \\ 0.8 \\ 10.0 \end{pmatrix} Position_{Plane} = \begin{pmatrix} 5.0 \\ -0.4 \\ 5.0 \end{pmatrix}$$

I've created two solutions for the code. The first one is just the code I created without looking at my code or any material. The seconds solution is a revamped and detailed solution. I would like to get feedback for both if possible.

First solution:

```
vertex_count = 6 * 2 * 3
for e in entities:
    matrix = (ModelMatrix.identity()
        .translate(e.position)
        .scale(e.scale))
    gl.glLoadUniform(self.transformation_matrix_loc, matrix.c_ptr())
    e.drawSolidCube()
```

Second solution:

```
for e in entities:
    self.model_matrix.load_identity()
    self.model_matrix.translate(e.get_position())
    self.model_matrix.scale(e.get_scale())

    self.shader.set_model_matrix(self.model_matrix.matrix)
    e.drawSolidCube()

    pygame.display.flip()
```

5.1.2 b) (5%) Show the values that are in the shader's model matrix when the first box is drawn.

$$model\_matrix = \begin{pmatrix} 2 & 0 & 0 & 9 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Check with python for validation:

```
>>> import glm
>>> mat = glm.mat4()
>>> mat = glm.translate(mat, e.position)
>>> mat = glm.scale(mat, e.scale)
>>> print(mat)

[           2 |           0 |           0 |           0 ]
[           0 |           2 |           0 |           0 ]
[           0 |           0 |           2 |           0 ]
[           9 |           5 |          -2 |           1 ]
```

LaTeX is so much fun :)

5.2 5. Camera Transformations (30%) (2015, No. 5)

5.2.1 a) (15%) A camera is set up to be positioned in (0,8,4) looking at the point (0,3,-1). It has an up vector (0,0,-1). Find the point of origin and vectors for the camera's coordinate frame.

$$origin = \begin{pmatrix} 0 \\ 8 \\ 4 \end{pmatrix}$$

I'm not a 100% sure which vectors I should calculate for the coordinate frame but I expect that these are $\vec{v}, \vec{n}, \vec{u}$ for the view matrix. This would also kind of make sense. So here we go:

$$\vec{n} = normalize(\vec{origin} - \vec{looking\_at}) = normalize \begin{pmatrix} 0-0 \\ 8-3 \\ 4--1 \end{pmatrix} = normalize \begin{pmatrix} 0 \\ 5 \\ 5 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0.707 \\ 0.707 \end{pmatrix}$$

$$\vec{u} = normalize\,(\vec{up} \times \vec{n}) \approx normalize \begin{pmatrix} 0.707 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\vec{v} = normalize\,((\vec{n} \times \vec{u})) \approx \begin{pmatrix} 0 \\ 0.707 \\ -0.707 \end{pmatrix}$$

5.2.2 (5%) Set up the values in a matrix that represents this position and orientation of a camera. Which matrix in your shader should be set to these values?

From formula sheet: $\begin{pmatrix} u_x & u_y & u_z & -(eye \circ u) \\ v_x & v_y & v_z & -(eye \circ v) \\ n_x & n_y & n_z & -(eye \circ n) \\ 0 & 0 & 0 & 1 \end{pmatrix}$

$$\texttt{ViewMatrix} \approx \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.707 & -0.707 & -2.828 \\ 0 & 0.707 & 0.707 & -8.485 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The view matrix should be set to these values.

5.2.3 (10%) The camera should have a field of view of 75 grad, an aspect ratio of 16:9, a near plane at 3 and a far plane at 25. Find the exact values for a matrix that calculates this camera. Which matrix in your shader should be set to these values?

From formula sheet: $\begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2*F*N}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix}$

N $= 3$

F $= 25$

fov $= 75$

aspect $= \frac{16}{9}$

top $= N * tan\left(\frac{fov}{2}\right) \approx 3 * 0.767 \approx 2.301$

bottom $= -top \approx -2.301$

right $= top * aspect \approx 4.091$

left $= -right \approx -4.091$

`ProjectionMatrix` $\approx \begin{pmatrix} 0.733 & 0 & 0 & 0 \\ 0 & 1.304 & 0 & 0 \\ 0 & 0 & -1.273 & -6.812 \\ 0 & 0 & -1 & 0 \end{pmatrix}$

The projection matrix should be set to these values.

6 TGRA Hand-in 1 (97/100)

6.1 3. Window-2-Viewport mapping (10%) (2014, No. 03)

6.1.1 Points are drawn in a 2D world window (l,r,b,t) = (-10, 30, 50, 80). In which pixels on a 1600x1200 viewport (bottom left corner (0,0)) will the following points be rendered?

1. P1 = (-5, 70)
2. P2 = (20, 65)

$$window_{origin} = \begin{pmatrix} -10 \\ 50 \end{pmatrix}$$

$$width_{window} = (30 - -10) = 40$$

$$height_{window} = (80 - 50) = 30$$

$$width_{viewport} = 1600$$

$$height_{viewport} = 1200$$

$$w\vec{2}v = \begin{pmatrix} \frac{width_{viewport}}{width_{window}} \\ \frac{height_{viewport}}{height_{window}} \end{pmatrix} = \begin{pmatrix} \frac{1600}{40} \\ \frac{1200}{30} \end{pmatrix} = \begin{pmatrix} 40 \\ 40 \end{pmatrix}$$

$$0\vec{P}1 = \begin{pmatrix} -5 \\ 70 \end{pmatrix}$$

$$0\vec{P}2 = \begin{pmatrix} 20 \\ 65 \end{pmatrix}$$

$$P1Pixel = \left(0\vec{P}1 - window_{origin}\right) * w\vec{2}v$$

$$P1Pixel = \left(\begin{pmatrix} -5 \\ 70 \end{pmatrix} - \begin{pmatrix} -10 \\ 50 \end{pmatrix}\right) * \begin{pmatrix} 40 \\ 40 \end{pmatrix}$$

$$P1Pixel = \begin{pmatrix} 5 \\ 20 \end{pmatrix} * \begin{pmatrix} 40 \\ 40 \end{pmatrix} = \begin{pmatrix} 200 \\ 800 \end{pmatrix}$$

$$P2Pixel = \left(0\vec{P}2 - window_{origin}\right) * w\vec{2}v$$

$$P2Pixel = \left(\begin{pmatrix} 20 \\ 65 \end{pmatrix} - \begin{pmatrix} -10 \\ 50 \end{pmatrix}\right) * \begin{pmatrix} 40 \\ 40 \end{pmatrix}$$

$$P2Pixel = \begin{pmatrix} 30 \\ 15 \end{pmatrix} * \begin{pmatrix} 40 \\ 40 \end{pmatrix} = \begin{pmatrix} 1200 \\ 600 \end{pmatrix}$$

P1 would be drawn at (200, 800) with a size of 40 x 40 px

P2 would be drawn at (1200, 600) with a size of 40 x 40 px

6.2 7. (10%) Vector intersections and reflections (2015, No. 7)

A line has end points (3,8) and (7,6). A particle starts at (4,2) and travels along in the direction (1,3).

6.2.1 a) (7%) In which point does the path of the particle cross the line?

The line can also be represented as a point and a direction like:

$L = \begin{pmatrix} 3 \\ 8 \end{pmatrix} + a * \left(\begin{pmatrix} 7 \\ 6 \end{pmatrix} - \begin{pmatrix} 3 \\ 8 \end{pmatrix} \right) = \begin{pmatrix} 3 \\ 8 \end{pmatrix} + a * \begin{pmatrix} 4 \\ -2 \end{pmatrix}$

The particle position and movement can be described in the same way:

$P = \begin{pmatrix} 4 \\ 2 \end{pmatrix} + b * \begin{pmatrix} 1 \\ 3 \end{pmatrix}$

We can now create a system of equations (Sorry I don't know the right english name)

$\begin{pmatrix} 3 \\ 8 \end{pmatrix} + a * \begin{pmatrix} 4 \\ -2 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix} + b * \begin{pmatrix} 1 \\ 3 \end{pmatrix} : a = \frac{9}{14}, b = \frac{11}{7}$

From this solution we can safely say that why collide because: 1. the equation has a solution 2. a is ¡ 1 and therefor between (3, 8) and (7, 6) The particle meets the line at: (5.57, 6.71)

$P\left(b = \frac{11}{7}\right) = \begin{pmatrix} 4 \\ 2 \end{pmatrix} + b * \begin{pmatrix} 1 \\ 3 \end{pmatrix} \approx \begin{pmatrix} 5.57 \\ 6.71 \end{pmatrix}$

This is maybe not the way it's done in games but it works here :)

6.2.2 b) (3%) If the particle is made to bounce off the line, what will it's new direction vector be?

a $= \begin{pmatrix} 1 \\ 3 \end{pmatrix}$

r $= a - 2\left(\frac{a \circ n}{n \circ n} * n\right)$

with $|n| = 1$:

r $= a - 2(a \circ n) * n$

$\vec{n} = \begin{pmatrix} 2 \\ 4 \end{pmatrix} * \frac{1}{\sqrt{2^2+4^2}}$ Note: $(|n| = 1)$

r $= \begin{pmatrix} 1 \\ 3 \end{pmatrix} - 2\left(\begin{pmatrix} 1 \\ 3 \end{pmatrix} \circ \begin{pmatrix} 2 \\ 4 \end{pmatrix} * \frac{1}{\sqrt{20}}\right) * \begin{pmatrix} 2 \\ 4 \end{pmatrix} * \frac{1}{\sqrt{20}}$

r $= \begin{pmatrix} 1 \\ 3 \end{pmatrix} - 2\left(\frac{14}{\sqrt{20}}\right) * \begin{pmatrix} 2 \\ 4 \end{pmatrix} * \frac{1}{\sqrt{20}}$

r $= \begin{pmatrix} 1 \\ 3 \end{pmatrix} - \frac{28}{\sqrt{20}} * \begin{pmatrix} 2 \\ 4 \end{pmatrix} * \frac{1}{\sqrt{20}}$

r $= \begin{pmatrix} 1 \\ 3 \end{pmatrix} - \frac{28}{20} * \begin{pmatrix} 2 \\ 4 \end{pmatrix}$

r $= \begin{pmatrix} -1.8 \\ -2.6 \end{pmatrix}$

7 Other formula

$d := distance$

$f := shininess$

$lambert = max\left(\frac{n \circ S}{|n| * |S|}, 0\right)$

$phong = max\left(\frac{n \circ h}{|n| * |h|}, 0\right)$

$attFactor = attenuation.x * d^2 + attenuation.y * d + attenuation.z$

$I = \sum \left(\left(I_d * \frac{material_d * lambert}{attFactor}\right) + \left(I_s * \frac{material_s * phong^f}{attFactor}\right)\right) + (I_{gaf} * material_a)$

$Reflection(\vec{a}) = \vec{a} - 2 * (\vec{a} \circ \vec{n}) * \vec{n}$