<table>
<tr><td colspan="3">Politechnika Świętokrzyska w Kielcach

Wydział Elektrotechniki, Automatyki i Informatyki

Katedra Informatyki, Elektroniki i Elektrotechniki</td></tr>
<tr><td>Kierunek

Informatyka</td><td colspan="2">Laboratorium

Podstawy grafiki komputerowej</td></tr>
<tr><td>Grupa dziekańska

3ID14A</td><td>Temat ćwiczenia

Laboratorium - Prototyp silnika 3D</td><td>Wykonali:

• Filip Borowiec</td></tr>
</table>

Laboratorium 7

Podobnie jak w przypadku sfml naszym pierwszym krokiem było utworzenie ekranu oraz zainicjowanie bibliotek, w naszym przypadku używamy bibliotekę glew do opengl, przygotowaliśmy takzę inncjalizacje pod różnego rodzaju obiekty które pojawią się w przyszłości, Dodaliśmy także pętle główną z ktorej będzie korzystać silnik

```
void mainLoop()
{


	unsigned long int timer;


	while(!this->shouldClose)
	{
		timer = clock();
		processInput();
		clear();


		this->updateShaders();
		this->render();
		this->update();

		glfwSwapBuffers(this->mainWindow);
		if (glfwWindowShouldClose(this->mainWindow))
		{
			this->shouldClose = true;
		}
		this->resetOneTimeEvenets();
		time running = clock();
		timeElapesed = timerunning - timer;
	}
}
```

Laboratorium 8

ze względu na to że obiekty w 3d są bardziej skomplikowane aniżeli w przestrzeni 2d musieliśmy się upewnić że mamy dobrze przystosowaną strukturę pod ich tworzenie oraz renderowanie, na samym początku stworzyliśmy wirtualną klasę renderable z której będą życzyły wszystkie klasy które ma nasz silnik renderować Laboratorium 9

```
class Gl Object :
          public Renderable,
          public Transformable
{
protected:


          std::shared_ptr<VAO> vao;
          std::shared_ptr<VBO> vbo;
          std::shared_ptr<Material> mat;

          int renderMode = GL_FILL;
          long int usage = GL_STATIC_DRAW;



          virtual int getVerteciesAmount() = 0;
          virtual std::shared_ptr<Attribute>
getVertexAttrib(int num, std::string attrib) = 0;


          void iniitVboUniversal(GLfloat*
vertexBuffer, int verteciesAmt, int vertexSize,
std::shared_ptr<VBO>& vbo, std::shared_ptr<VAO>&
vao)
          {
            /*
            for (int i = 0; i < verteciesAmt *
vertexSize; i++)
            {
                      std::cout << vertexBuffer[i]
<< ",";
                      if (i % vertexSize ==
vertexSize - 1)
                                std::cout <<
std::endl;
            }
            */

            vao->bind();
            vbo = std::make_unique
<VBO>(vertexBuffer, this->getVerteciesAmount() *
this->getVertexSize() * sizeof(float), usage);
            delete[] vertexBuffer;
          }

          void iniitVao()
          {

            int verteciesAmt =
this->getVerteciesAmount();
            int vertexSize = this->getVertexSize();
            std::map<std::string, Attribute::Types>
attributesMap = mat->getAttributeMap();
            std::list<std::string> attributeList =
mat->getAttributeList();

            int i = 0;
            int offset = 0;

            for (std::string name : attributeList)
            {
                      Attribute::Types type =
attributesMap[name];
                      this->vao->linkAttrib(vbo, i,
type, GL_FLOAT, vertexSize * sizeof(float),
offset);
                      glEnableVertexAttribArray(i);
                      //std::cout <<
"glEnableVertexAttribArray(" << i << ")\n";
                      i++;
                      offset += type *
sizeof(float);
            }

          }

          virtual void bind()
          {
            this->vbo->bind();
            this->vao->bind();
          }
          virtual void unbind()
          {
            vao->unbind();
            vbo->unbind();
          }
          virtual void glDrawCall() = 0;

          void renderProc() override
          {
            glPolygonMode(GL_FRONT_AND_BACK,
renderMode);
            if (mat != nullptr)

mat->apply(this->getTransformationMatrix());
            bind();
            glDrawCall();
            unbind();
            mat->unapply();
          }

};
```

Podobnie jak w przypadku transformacji wykorzystaliśmy bibliotekę glm do stworzenia macierzy obsługująca kamerę. Stworzyliśmy klasę do obsługi kamery.

```
#pragma once
#include <complex>

#include "Mouse.h"
#include "Transformable.h"
class Camera :
	public Transformable
{
private:

	glm::mat4 projection = glm::mat4(1);

	float fov;
	float aspect;
	float nearPlane;
	float farPlane;
	const float cameraSpeedFactor =0.2;

	void iniit()
	{
	projection = glm::perspective(glm
::radians( this->fov), this->aspect,
this->nearPlane, this->farPlane);
	}

	glm::mat4 getLookAtMatrix()
	{
	return
glm::lookAt(glm::vec3(this->getFullPositon().x,
this->getFullPositon().y,
this->getFullPositon().z), glm::vec3(0, 0, 0),
glm::vec3(0, 1, 0));
	}


public:
	Camera(float fov, float aspect, float
near_plane, float far_plane)
	: fov(fov),
	aspect(aspect),
	nearPlane(near_plane),
	farPlane(far_plane),
	Transformable()
	{
	iniit();
	}

	float get_fov() const
	{
	return fov;
	}

	void set_fov(float fov)
	{
	this->fov = fov;
	iniit();
	}
```

```
	float get_aspect() const
	{
	return aspect;
	}

	void set_aspect(float aspect)
	{
	this->aspect = aspect;
	iniit();
	}

	float get_near_plane() const
	{
	return nearPlane;
	}

	void set_near_plane(float near_plane)
	{
	nearPlane = near_plane;
	iniit();
	}

	float get_far_plane() const
	{
	return farPlane;
	}

	void set_far_plane(float far_plane)
	{
	farPlane = far_plane;
	iniit();
	}



	glm::mat4 getCameraMatrix()
	{
	return
projection*this->getLookAtMatrix() ;
	}

	void set_aspect(int width, int height)
	{
	if(width>0 && height>0)
	set_aspect((float)width / (float)height);
	}

	void applyZoom(float value)
	{
	//
this->moveIndendent(getFullPositon()*value);
	//  std::cout << this->get_position() <<
std::endl;
	Vector3f scaling = Vector3f(1, 1, 1) ;
	if (value != 0) {
	if(value>0)
	scaling = Vector3f(0.9, 0.9, 0.0) ;
	else
```

```
        scaling = Vector3f(1.1, 1.1, 1.1) ;
        scale(scaling);
        }
        // move(pos);
        //std::cout << getFullPositon() << ", "
<< get_position() <<", "<<value<< std::endl;
        }



        void applyMouseMovement(Vector2f
movement)
        {


        Vector3f Pos= getFullPositon() -
get_position();
```

```
        glm::mat4 yawRotation =
glm::rotate(glm::mat4(1.0f),
glm::radians(movement.x) * cameraSpeedFactor,
glm::vec3(0, 1, 0));
        glm::mat4 pitchRotation =
glm::rotate(glm::mat4(1.0f),
glm::radians(movement.y*cameraSpeedFactor),
glm::cross(glm::vec3(0, 1, 0), Pos.glm()));
        originPointTransformation = yawRotation*
pitchRotation* originPointTransformation;


        }
};
```

następnie macierzy perpskety kamery jest w naszym silniku przenoszona do shaderów które za mnożenia macierzy decydują o pozycji obiektu na ekranie

Laboratorium 10

Aby zaimportować cieniowanie potrzebowaliśmy aby nasz vertex buffer zawierał parametry normal czyli kierunek wskazywania danego punktu aby móc określić jak światło pada na daną powierzchnię, po pierwsze musieliśmy stworzyć klasę shader która będzie informować obiekt renderowany tym shader że potrzebuje wartości normalnych

```
class ShaderProgram
{
    unsigned int ID;
    std::map<std::string, Attribute::Types>
AttributesRequiredMap;
    std::list<std::string>
AttributesRequiredList;
    int vertexSize = 0;
    std::string getTextFile(std::string
file);


public:


    ShaderProgram(std::string
vertexShaderSource, std::string
fragmentShaderSource);
    void use();
    void setBool(const std::string& name,
bool val);
    void setInt(const std::string& name, int
val);
    void setFloat(const std::string& name,
float val);
```

```
    void setArray(const std::string& name,
int size, GLfloat* pointer);
    void unuse();
    void setVector3f(const std::string&
name, const Vector3f& vector3_f);
    void setMatrix4(const std::string& name,
const glm::mat4& mat4);
    void addAttribute(std::string atribName,
Attribute::Types type);
    std::list<std::string>&
getAttributeList();
    std::map<std::string, Attribute::Types>&
getAttributeMap();
    unsigned int getVertexSizeRequired();
    void setCamera(std::string  name,
Camera* camera);
    bool isLightAffeted();
    void applyLightData(Vector3f cameraPos);
};
```

dodatkowo aby można było zaużywać efekty tego potrzbujemy także infromacj io naszym oświetleniu, Utworzlyiśmy strukturę naszego oświelenie odziweciadlajća strukture danych w naszym sahderze zawierający dane jak pozycje i parametry światła

```
#define  MAXLIGHTS 9

class Light
{
protected:

    enum LIGHT TYPE
    {
         POINYLIGHT,
         DIRECTIONALLIGHT,
         SPOTLIHGT
    };



    struct LightStructType
    {
         LIGHTYTYPE type;
         float intensity;
         Vector3f color;
         Vector3f position;
         Vector3f  direction;
         float angle;

    };



    char lightNum;
```

```
    struct lightData
    {
         int light Counter;
         float ambientIntensity;
         Vector3f ambientColor;
         LightStructType lightTypeData[MAX
LIGHTS];
    };



public:
    static lightData light_data;

    Light();

    static Float getAmbientIntensity();

    static void setAmbientIntensity(float
val);

    static Vector3f getAmbientColor();
    static void setAmbientColor(Vector3f
col);

    void setLightColor(Vector3f color);

    Vector3f getLightColor();

    void setLightIntensity(float intensity);
    Float getLightIntensity();

    void setLightType(LIGHTYTYPE type);

    LIGHTYTYPE getLightType();

    void applyLight(ShaderProgram* shader);
```

};

póki co zaimplementowana jest tylko światło punktowe ale jest przygotowane pod wystapenie także innych typów oświetlaenia, Natomiast sama klasa pointl l;ight także dziedziczy po polygonal abyśmy mogli zobaczyć skąd pada światło

```
class Point Light :
          public Light,
          public Cube
{
private:
          void setLightPos()
          {

light_data.lightTypeData[lightNum].position =
getFullPositon();

          }

public:

          PointLight(): Light(), Cube(1)
          {

this->setMat(std::make_shared<Material>(ShaderLib::
```

```
guideShader));
          init();

          setLightType(POINYLIGHT);
          this->set_render_mode(GL_LINE);
          }
          void move(Vector3f offset) override
          {
          Transformable::move(offset);
          setLightPos();
          }
          void set_position(Vector3f position)
override
          {
          Transformable::set_position(position);
          setLightPos();
          }
};
```

a tak wygladją nasze obliczenia oświetlenia

```
ersion 330 core
#define MAXLIGHTS 9

out vec4 FragColor;
in vec2 Uv;
in vec3 N;
in vec3 P;
uniform vec3 eyeP;
uniform sampler2D colorTexture;
uniform sampler2D normalTexture;
uniform sampler2D roughTexture;
uniform sampler2D emissionTexture;


    struct LightStructType
    {
          int type;
          float intensity;
          vec3 color;
          vec3 position;
          vec3  direction;
          float angle;

    };




    struct lightData
    {
          int amtofLight;
          float ambientIntensity;
          vec3 ambientColor;
          LightStructType
lightTypeData[MAXLIGHTS];
    };

    uniform lightData light_data;
```

```
void main()
{


    vec3 objectColor =vec3( texture(colorTexture,
Uv));
    vec3 objectNormal = abs(vec3(
texture(normalTexture, Uv).xyz*2-1)-1);
    vec3 objectRough =abs( vec3(
texture(roughTexture, Uv))-1);

    vec3
ambientLight=light_data.ambientColor*light_data.amb
ientIntensity;

    vec3 norm = normalize(N);

    vec3 lightDiff=vec3(0);

    vec3 lightDir =
normalize(light_data.lightTypeData[0].position -
P);
    float lightVal=max(dot(norm,lightDir),0);
    vec3
diffuse=lightVal*light_data.lightTypeData[0].color;
    lightDiff+=diffuse;

    vec3 viewDir = normalize(eyeP-P);
    vec3 reflectDir=reflect(-lightDir,norm);
    float
spec=pow(max(dot(viewDir,reflectDir),0.0),32);
    vec3
specular=light_data.lightTypeData[0].color*(spec);
    vec3
finalColor=vec3((ambientLight+lightDiff)*objectColo
```

```
r+objectRough.r*specular);
         FragColor=vec4(finalColor,1);

 //FragColor=vec4(objectRough ,1);
```

```
}
```

Laboratorium 11

aby zaimportować dodawanie tektur po pierwsze musimy dodać możliwość ich dodawania w shaderze, (uwzględnienie w poprzednim listingu)

aby załadować tekstury stworzyliśmy kalse obsługująca tekstury z wykorzystaniem biblioteki stb image

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
void Texture::genData(unsigned char* data, int
width, int height, int channels, int format)
{
    glGenTextures(1, &ID);
    glBindTexture(GL_TEXTURE_2D, ID);
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width,
height, 0, format, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}

void Texture::genTex(std::string file, int width,
int height, int channels, int format)
{
    stbi_set_flip_vertically_on_load(true);
    unsigned char* data = stbi_load(file.c_str(),
&width, &height, &channels, 0);
    if (data == nullptr)
    {
```

```
        throw std::runtime_error("couldnt load
photo");
    }

    genData(data, width, height, channels, format);
    stbi_image_free(data);
}

Texture::Texture(std::string file, int width, int
height, int channels, int format)
    : width(width),
    height(height),
    channels(channels)
{
    genTex(file, width, height, channels, format);
}



void Texture::bind()
{
    glBindTexture(GL_TEXTURE_2D, ID);
}
void Texture::unbind()
{
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

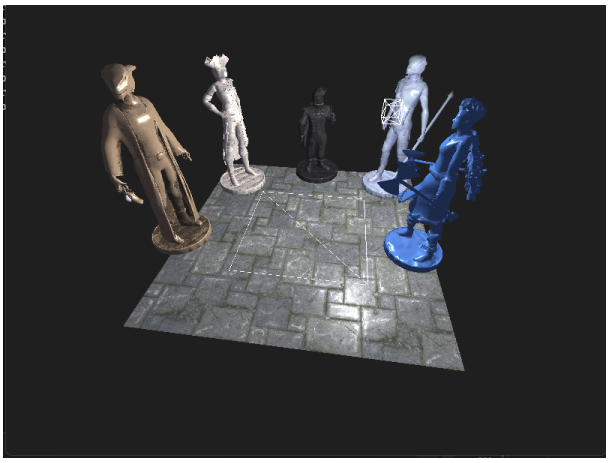
dodalismy takzę że zamiast pojedyczej kalsy obśługjącej sahder doaliśmy klase materii która zawiera shader ale także domyślne parametry do neigo i tekstury do załączenia

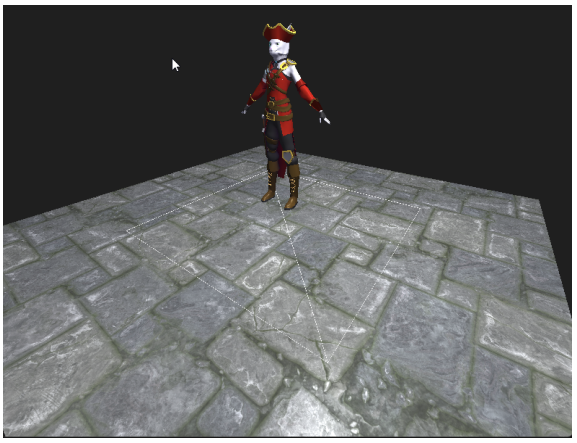
```
class Material : public Object
{
```

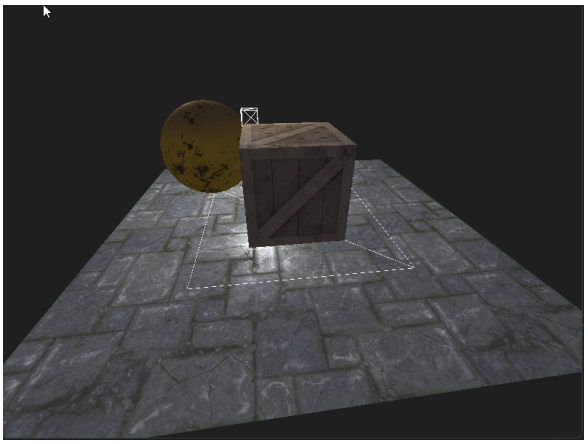
```
protected:
    ShaderProgram* shader;
```

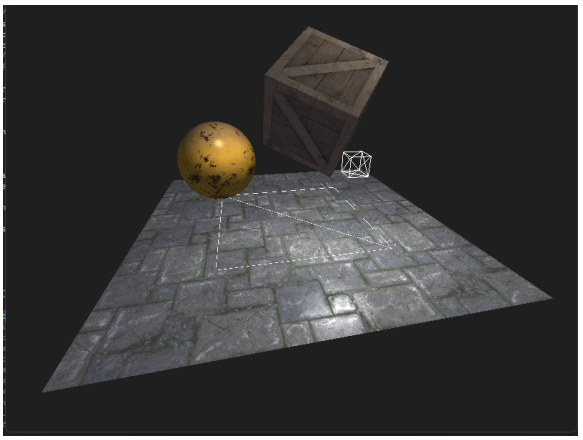
```
    std::vector< std::shared_ptr<Texture>>
textures;


    virtual void bindTex()
    {
          int i = 0;
          for ( std::shared_ptr<Texture> tex :
textures)
          {
                glActiveTexture(GL_TEXTURE0 +
i);
                tex->bind();
                i++;
          }
    }
    void unBindTex()
    {
          int i = 0;
          for ( std::shared_ptr<Texture> tex :
textures)
          {
                glActiveTexture(GL_TEXTURE0 +
i);
                Texture::unbind();
                i++;
          }
    }

public:
    explicit Material(ShaderProgram* shader)
          : shader(shader)
    {
          if (shader == nullptr)
          {
                throw
std::runtime_error("cannot create materal with null
shader");
          }
    }

    void addTex(std::string name,
std::shared_ptr<Texture> tex)
    {
          shader->use();
          shader->setInt("colorTexture",
textures.size());
          textures.push_back(tex);

          shader->unuse();
    }



    void apply(glm::mat4 model)
    {

          shader->use();
          shader->setMatrix4("model", model);
          bindTex();
    }



    void unapply()
    {
          unBindTex();
          shader->unuse();
    }

    std::list<std::string> getAttributeList()
    {
          return shader->getAttributeList();
    }

    std::map<std::string, Attribute::Types>
getAttributeMap()
    {
          return shader->getAttributeMap();
    }

    int getVertexSizeRequired()
          {
          return shader->getVertexSizeRequired();
          };
};
```

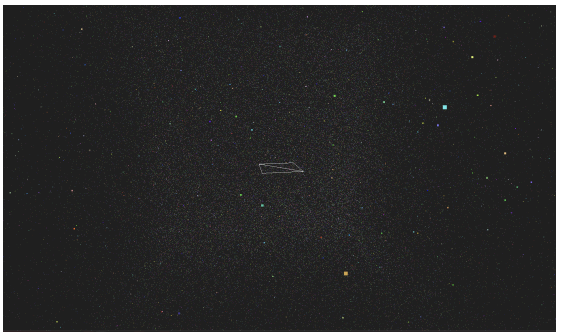
Screeny z możliwości silnika

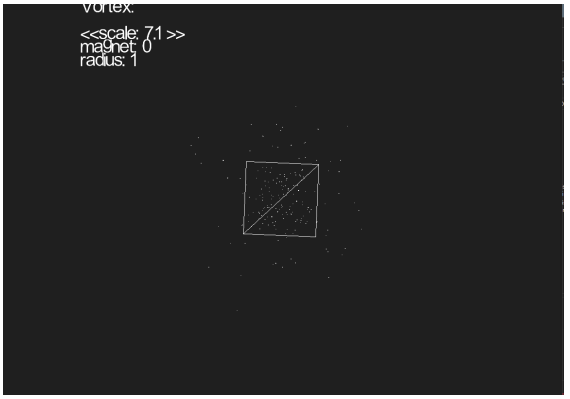











Vortex:
<<scale: 7.1 >>
magnet: 0
radius: 1


Vortex:
scale: 7.1
<<magnet: 200 >>
radius: 1