МИНОБРНАУКИ РОССИИ
РГУ НЕФТИ И ГАЗА (НИУ) ИМЕНИ И.М. ГУБКИНА

Факультет Автоматики и Вычислительной Техники

Кафедра Автоматизированных систем управления

Оценка комиссии: Рейтинг:

Подписи членов комиссии:

(подпись) (фамилия, имя, отчество)

(подпись) (фамилия, имя, отчество)

(дата)

КУРСОВАЯ РАБОТА

по дисциплине Компьютерная графика

на тему Разработка графического приложения с использованием современных графических API

«К ЗАЩИТЕ»

Доцент к.т.н. Папилина Т. М.

(должность, ученая степень; фамилия, и.о.)

(подпись)

(дата)

ВЫПОЛНИЛ :

Студент группы **АС-18-04**

(номер группы)

Ломакин Даниил Дмитриевич

(фамилия, имя, отчество)

(подпись)

(дата)

Москва, 2021

МИНОБРНАУКИ РОССИИ

РГУ НЕФТИ И ГАЗА (НИУ) ИМЕНИ И.М. ГУБКИНА

Факультет **Автоматики и вычислительной техники**
Кафедра Автоматизированных систем управления

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

по дисциплине Компьютерная графика

на тему Разработка графического приложения с использованием современных графических API

ДАНО студенту **Ломакину Даниилу Дмитриевичу** (фамилия, имя, отчество в дательном падеже) группы **АС-18-04** (номер группы)

Содержание работы:

1. Изучить процесс разработки приложений с использованием современных графических API
2. Разработать графическое приложение в соответствии с требованиями к функциональности
3.

Исходные данные для выполнения работы:

1. Индивидуальны требования к функциональности графического приложения курсовой работы
2. Лабораторные работы по курсу

Рекомендуемая литература:

1. Спецификация OpenGL: https://www.khronos.org/registry/OpenGL/specs/gl
2. Подборка материалов по комп. графике https://www.realtimerendering.com/

Графическая часть:

1.
2.

Руководитель: к.т.н. (уч.степень) доцент (должность) _____ (подпись) Папилина Т.М. (фамилия, имя, отчество)

Задание принял к исполнению: студент _____ (подпись) Ломакин Д.Д (фамилия, имя, отчество)

Содержание

Введение

Forward rendering представляет собой простой подход, позволяющий рисовать каждый объект, учитывая все световые источники и вычисляя свет для каждого пикселя. Главным минусом прямого освещения является производительность, которая снижается из-за перебора всех световых источников для модели. При сложной сцене со множеством объектов и ситуаций, когда один и тот же пиксель покрывают несколько объектов, у нас будут пустые затраты ресурсов. Другая проблема заключается в неэффективном использовании данного подхода на сценах с большим количеством источников света, поскольку большая часть вычислений шейдера окажется ненужной и будет перезаписана значениями для более близких объектов.

Отложенное освещение позволяет решить данные проблемы. Главная идея подхода-разделение алгоритма на 2 части: в 1 проходе рисуется вся сцена, различная информация сохраняется в набор текстур (G-буфер) для каждого пикселя, во 2 проходе мы используем данный набор текстур при рисовании всей сцены и расчете освещения. Расчет освещения остается таким же, как и при прямом освещении, но с использование g-буфера (рисунок 1).

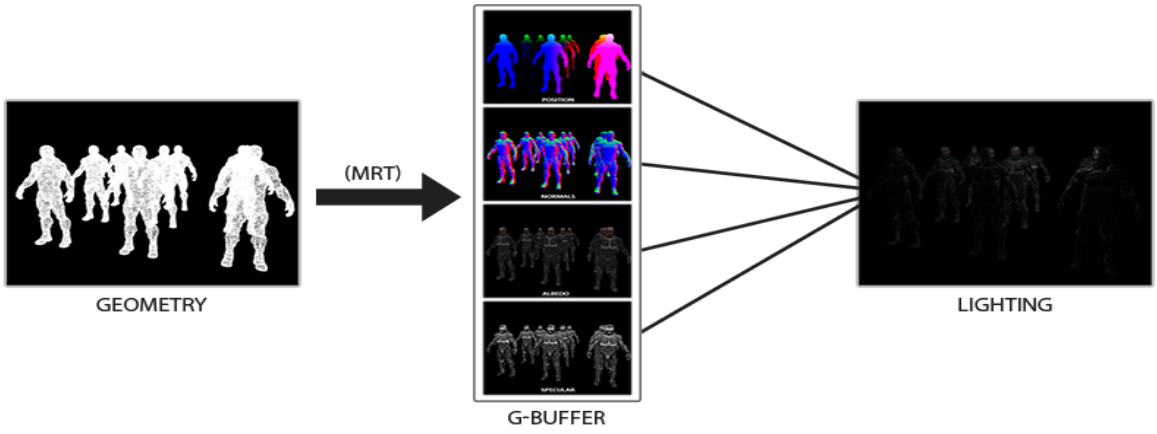

Рисунок 1-Алгоритм отложенного рендеринга

Постановка задачи

В данной работе мы должны рассмотреть один из алгоритмов отложенного освещения и показать, как он работает на фигурах. Необходимо прописать

алгоритм, создать шейдеры и необходимые модели фигур, реализовать его применение с прямым рендерингом.

Модель Блинна-Фонга

Модель освещения Блинна-Фонга (рисунок 2) используется во многих графических движках. Благодаря данной модели можно получить более реалистичную картинку.

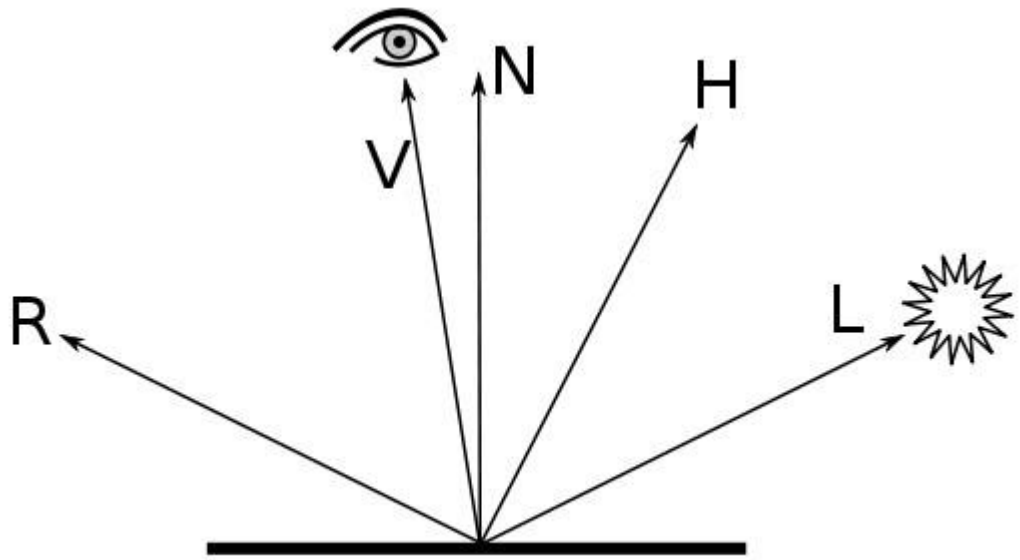

Рисунок 2 - Модель освещения Блинна-Фонга

N – нормаль к поверхности

L – направление к источнику света

R – направление отраженного луча

V – направление к наблюдателю

В модели Блинна - Фонга вводится вектор H, который является «медианой» угла между V и L. Вектор H вычисляется по формуле:

$$H = \frac{L+V}{|L+V|}$$

Таким образом, итоговая формула для модели освещения Блинна - Фонга имеет следующий вид: $I_s = k_s \cos^{\alpha}(\vec{N},\vec{H}) i_s$.

G-буфер

G-буфером подразумевает текстуры, которые мы будем использовать для сохранения связанной с освещением информации, используемой во втором проходе рендеринга. При прямом рендеринге для расчета освещения по модели Блинна-Фонга мы используем следующие переменные:

- 3-d вектор позиции.
- Дуффузный цвет фрагмента (отражательная способность для красного, зелёного и синего цветов).
- 3-d вектор нормали.
- Float для хранения зеркальной составляющей.
- Позиция источника света и его цвет.
- Позиция камеры.

Вектор позиции используется, чтобы узнать положение фрагмента относительно позиций камеры и источников света. Вектор нормали требуется для расчета угла падения света на поверхность.

В алгоритме отложенного освещения мы передадим данные в финальный проход. Мы получим тот же самый результат, несмотря на то, что мы будет рисовать фрагменты на обычном 2-d прямоугольнике. Необходимо хранить всю информацию в текстурах размером с экран (G-буфере) и использовать ее в проходе освещения. Мы получим такие же входные данные, как и при прямом освещении, при совпадении размера текстур и экрана.

Геометрический проход

Для геометрического прохода создадим фреймбуфер с очевидным именем gBuffer, к которому присоединим несколько цветовых буферов и один буфер глубины (рисунок 3). Для хранения позиций и нормали предпочтительно использовать текстуру с высокой точностью (16 или 32-битные float значения для каждой компоненты), диффузный цвет и значения зеркального отражения мы будем хранить в текстуре (точность 8 бит на компоненту). Так как мы используем несколько целей рендеринга, мы должны явно указать OpenGL, в какие буферы из присоединённых к GBuffer мы собираемся рисовать в glDrawBuffers() [1].

```
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
unsigned int gPosition, gNormal, gAlbedoSpec;
// Цветовой буфер позиций
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, gPosition, 0);

// Цветовой буфер нормалей
glGenTextures(1, &gNormal);
glBindTexture(GL_TEXTURE_2D, gNormal);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, gNormal, 0);

// Цветовой буфер значений цвета + отраженной составляющей
glGenTextures(1, &gAlbedoSpec);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, gAlbedoSpec, 0);

//Указываем OpenGL на то, в какой прикрепленный цветовой буфер(заданного фреймбуфера) мы собираемся выполнять рендеринг
unsigned int attachments[3] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, attachments);

// Создаем и прикрепляем буфер глубины (рендербуфер)
unsigned int rboDepth;
glGenRenderbuffers(1, &rboDepth);
glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, SCR_WIDTH, SCR_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboDepth);

// Проверяем готовность фреймбуфера
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "Framebuffer not complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Рисунок 3 - Реализация буферов в коде

Каждый объект имеет цвет нормаль и коэффициент зеркального отражения. Нам необходимо отрендерить данные в g-буфере, поэтому шейдер (рисунок 4):

```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;

uniform sampler2D texture_diffuse1;
uniform sampler2D texture_specular1;

void main()
{
    //Храним вектор позиции фрагмента в первой текстуре g-буфера
    gPosition = FragPos;
//Также храним нормали каждого фрагмента в g-буфере
    gNormal = normalize(Normal);
    //И диффузную составляющую цвета каждого фрагмента
    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
    //Сохраним значение интенсивности отраженной составляющей в альфа-компоненте переменной gAlbedoSpec
    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

Рисунок 2 - Реализация шейдера для текстуры g-буфера

Освещение

В G-буфере у нас есть необходимая информация, поэтому мы имеем возможность полностью вычислить освещение и финальные цвета для каждого пикселя G-буфера, используя его содержание в качестве входных данных для алгоритмов расчёта освещения. Так как значения G-буфера представляют только видимые фрагменты, мы выполним сложные расчёты освещения ровно по одному разу для каждого пикселя. Благодаря этому отложенное освещение довольно эффективно, особенно в сложных сценах, в которых при прямом рендеринге для каждого пикселя довольно часто приходится производить вычисление освещения по нескольку раз [1].

Для данного этапа мы рендерим полноэкранный прямоугольник и производим вычисления освещения для каждого пикселя.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0); glBindTexture(GL_TEXTURE_2D,
gPosition); glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2); glBindTexture(GL_TEXTURE_2D,
gAlbedoSpec);
```

```
// и ещё в юниформы записываем информацию об освещении
shaderLightingPass.use();
SendAllLightUniformsToShader(shaderLightingPass);
shaderLightingPass.setVec3("viewPos", camera.Position); RenderQuad();
```

Первоначальный вид шейдера для освещения:

```
#version 330 core out vec4
FragColor; in vec2 TexCoords;
uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;
```

```
struct Light {     vec3 Position;
vec3 Color;
};
const int NR_LIGHTS = 32; uniform
Light lights[NR_LIGHTS]; uniform
vec3 viewPos;

void main()
{
   // получаем информацию из G-буфера     vec3
FragPos = texture(gPosition, TexCoords).rgb;     vec3
Normal = texture(gNormal, TexCoords).rgb;     vec3
Albedo = texture(gAlbedoSpec, TexCoords).rgb;     float
Specular = texture(gAlbedoSpec, TexCoords).a;

   // вычисляем освещение как обычно     vec3 lighting = Albedo
* 0.1; // хардкодим фоновое освещение     vec3 viewDir =
normalize(viewPos - FragPos);     for(int i = 0; i < NR_LIGHTS;
++i)
   {
      // рассеянное освещение         vec3 lightDir =
normalize(lights[i].Position - FragPos);         vec3 diffuse = max(dot(Normal,
lightDir), 0.0) * Albedo * lights[i].Color;         lighting += diffuse;
   }

   FragColor = vec4(lighting, 1.0);
}
```

Рисунок 3 - Результат, полученный при отложенном рендеринге

Шейдер освещения будет принимать 3 текстуры, которые содержат всю информацию, записанную в геометрическом проходе и из которых состоит Gбуфер. Значения будут совпадать со значениями прямого рендеринга. Сначала мы получаем значения, которые относятся к освещению, потом цвет и коэффициент зеркального отражения из gAlbedoSpec. Затем мы используем модель Блинна-Фонга для необходимых расчетов. Недостатки данного подхода заключаются в невозможности смешивания и использования одного общего для всех объектов способа расчета освещения.

Чтобы справиться с этими недостатками, комбинируют прямой и отложенный рендеринг. Для иллюстрации работы нарисуем источники в виде кубиков и сфер с помощью прямого рендеринга.

Комбинация отложенного рендеринга с прямым

Сначала нам нужно скопировать информацию о глубине из геометрического прохода в буфер глубины, и только после этого нарисовать светящиеся фигуры. Таким образом, фрагменты светящихся фигур будут нарисованы только в том случае, если они находятся ближе, чем уже нарисованные объекты [1].

Далее рендерим источники освещения вверху сцены (рисунок 6).

```
// 2.5. Копируем содержимое буфера глубины (геометрический проход) в буфер глубины заданного по умолчанию фреймбуфера
glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); // пишем в заданный по умолчанию фреймбуфер
glBlitFramebuffer(0, 0, SCR_WIDTH, SCR_HEIGHT, 0, 0, SCR_WIDTH, SCR_HEIGHT, GL_DEPTH_BUFFER_BIT, GL_NEAREST);
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 3. Рендерим источники освещения вверху сцены
shaderLight.use();
shaderLight.setMat4("projection", projection);
shaderLight.setMat4("view", view);
for (unsigned int i = 0; i < lightPositions.size() / 2; i++)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, lightPositions[i]);
    model = glm::scale(model, glm::vec3(0.125f));
    shaderLight.setMat4("model", model);
    shaderLight.setVec3("lightColor", lightColors[i]);
    renderSphere();
}
for (unsigned int i = lightPositions.size() / 2; i < lightPositions.size(); i++)
{
    model = glm::mat4(1.0f);
    model = glm::translate(model, lightPositions[i]);
    model = glm::scale(model, glm::vec3(0.125f));
    shaderLight.setMat4("model", model);
    shaderLight.setVec3("lightColor", lightColors[i]);
    renderCube();
}
```

Рисунок 4 - Реализация кода для совместного использования отложенного и прямого освещения

Далее необходимо узнать область действия источника, ищем размер. Мы должны найти область, в которой свет способен достигнуть поверхности. Так как большинство источников света используют какое-нибудь затухание, мы можем найти радиус, которое свет может достигнуть. После этого мы выполняем сложные расчёты освещения только для тех источников света, которые влияют на данный фрагмент.

Мы используем уравнение затухания, чтобы найти радиус действий:

$$\frac{5}{256} = \frac{I_{max}}{Attenuation}$$

Выбранная функция затухания становится практически тёмной на расстоянии радиуса действия, если мы ограничим её на меньшей яркости чем 5/256, то область действия источника света станет слишком большой — это не так эффективно. [1]

Радиус действия источника:

$$x = \frac{-K_l + \sqrt{K_l^2 - 4K_q(K_c - I_m ax\frac{256}{5})}}{2K_q}$$

float constant = 1.0; float linear   = 0.7; float quadratic = 1.8; float lightMax = std::fmaxf(std::fmaxf(lightColor.r, lightColor.g), lightColor.b); float radius   = (-linear + std::sqrtf(linear * linear - 4 * quadratic * (constant - (256.0 / 5.0) * lightMax))) / (2 * quadratic);

Шейдер для освещения:

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;
struct Light {
    vec3 Position;
    vec3 Color;
    float Linear;
    float Quadratic;
    float Radius;
};
const int NR_LIGHTS = 60;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;
void main()
{
    // Получаем данные из g-буффера
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Diffuse = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;
    // Затем вычисляем освещение как обычно
    vec3 lighting  = Diffuse * 0.1; // фоновая составляющая
    vec3 viewDir  = normalize(viewPos - FragPos);
    for(int i = 0; i < NR_LIGHTS; ++i)
    {
        // Вычисляем расстояние между источником света и текущим фрагментом
        float distance = length(lights[i].Position - FragPos);
        if(distance < lights[i].Radius)
        {
            // Диффузная составляющая
            vec3 lightDir = normalize(lights[i].Position - FragPos);
            vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * lights[i].Color;
// Отраженная составляющая
            vec3 halfwayDir = normalize(lightDir + viewDir);
            float spec = pow(max(dot(Normal, halfwayDir), 0.0), 16.0);
            vec3 specular = lights[i].Color * spec * Specular;
// Затухание
            float attenuation = 1.0 / (1.0 + lights[i].Linear * distance + lights[i].Quadratic * distance * distance);
            diffuse *= attenuation;
            specular *= attenuation;
            lighting += diffuse + specular;
        }
    }
    FragColor = vec4(lighting, 1.0);
}
```

Рисунок 5- Шейдер для реализации освещения при совместном использовании двух подходов

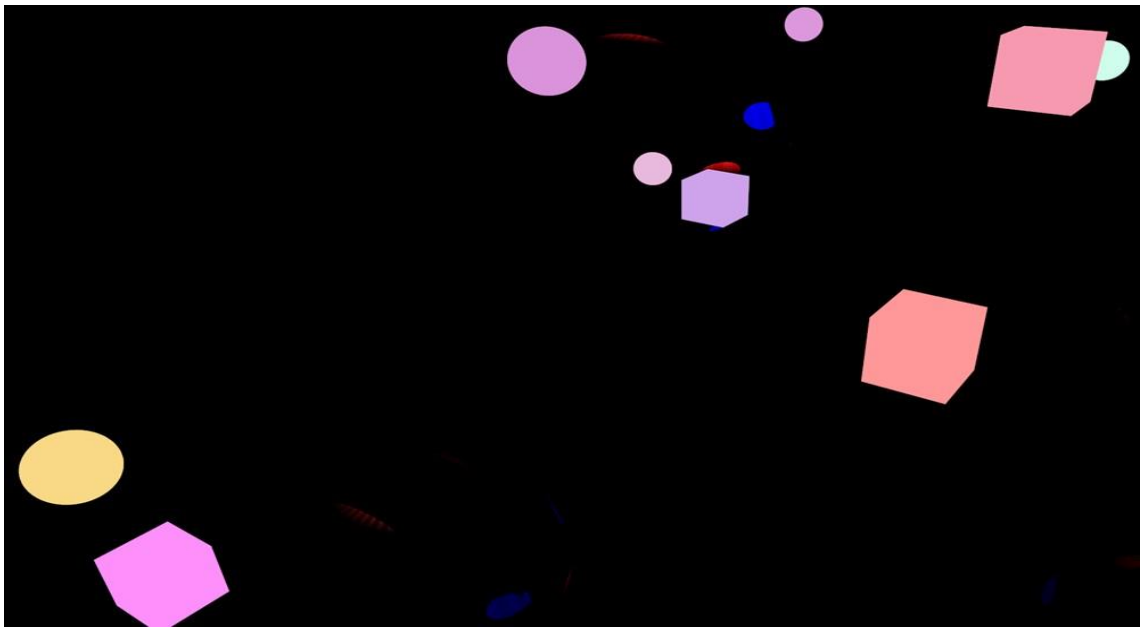

Рисунок 6 - Результат работы отложенного рендеринга совместно с прямым

Заключение

Поставленная задача считается выполненной: был прописан алгоритм отложенного освещения, созданы шейдеры и фигуры, также была показана работа алгоритма на фигурах. Отложенный рендеринг был рассмотрен и реализован, как и его сочетание с прямым рендерингом. Отложенный рендеринг хорошо функционирует как с прямым освещением, так и без него. Данные подходы были рассмотрены на сферах.

Источники литературы:

1. URL: https://habr.com/ru/post/420565
2. URL: https://triplepointfive.github.io/ogltutor/tutorials/tutorial35.html
3. Спецификация OpenGL: https://www.khronos.org/registry/OpenGL/specs/gl
4. Подборка материалов по комп. Графике: https://www.realtimerendering.com
5. Спецификации Vulkan: https://www.khronos.org/registry/vulkan
6. Шейдеры в OpenGL: https://ravesli.com/urok-5-shejdery-v-opengl