Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Прикладной математики

Лабораторная работа №3
по дисциплине «Комьютерная графика»

Трассировка лучей

Факультет: ПМИ

Группа: ПМ-63

Студент: Шепрут И.И.

Вариант: -

Преподаватель: Задорожный А.Г.

Новосибирск
2019

1 Цель работы

Ознакомиться с основными аспектами метода трассировки лучей.

2 Постановка задачи

1. Считывать из файла (в зависимости от варианта):
1.1. Тип объекта.
1.2. Координаты и размер объектов.
1.3. Параметры материала объектов.
2. Выполнить трассировку первичных лучей.
3. Добавить зеркальную плоскость и учесть отраженные лучи.
4. Предусмотреть возможность включения/исключения объектов.
5. Предусмотреть возможонсть изменения положения источника света.

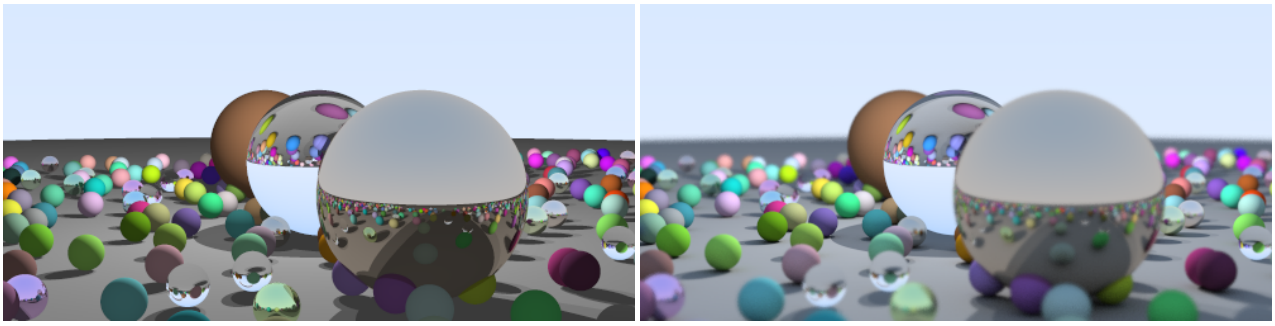
3 Реализованные функции

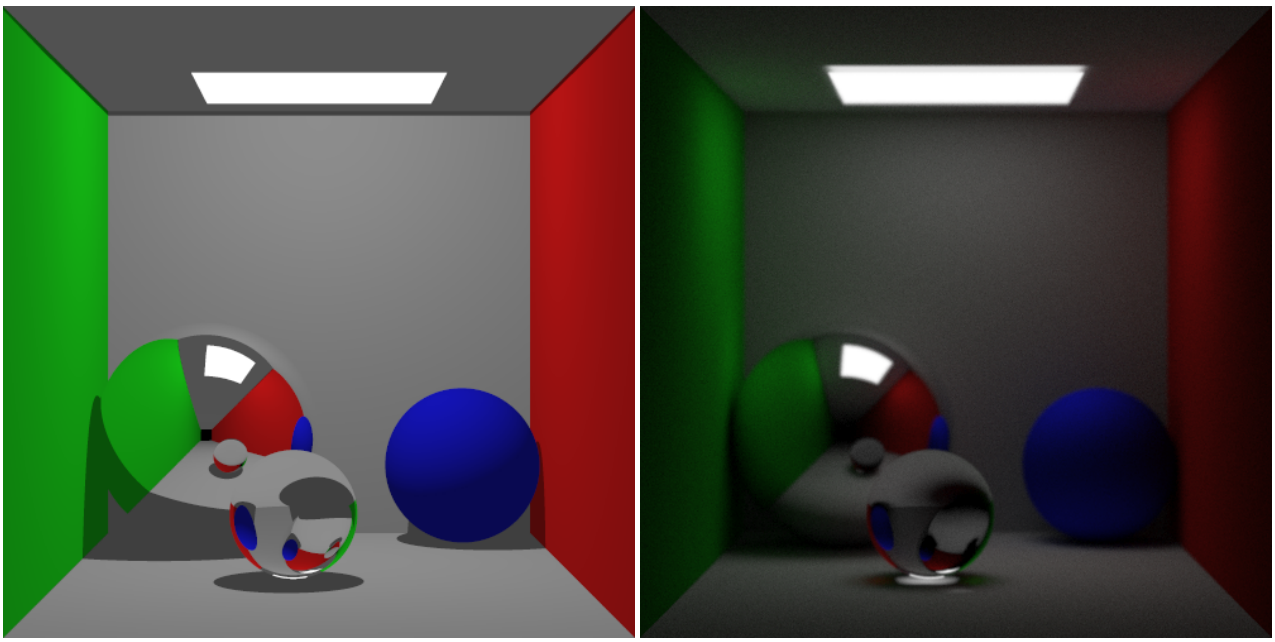
- Полиморфизм.
  - Различные подходы к рендерингу. — одну и ту же сцену можно рендерить различными классами и получать соответственно различную скорость и фотореалистичность.
    - Ray Tracing — обычная трассировка лучей, которая может отобразить картинку с малым количеством семплов. Трейсинг лучей останавливается при достижении материала с цветом, и рендерит вторичные лучи, только если попадает на преломляющую или зеркальную поверхность.
    - Path Tracing — методика рендеринга в компьютерной графике, которая стремится симулировать физическое поведения света настолько близко к реальному, насколько это возможно. Трейсинг лучей останавливается только при достижении предела отражений, или при достижении источника света.
  - Полиморфизм объектов — имеется два типа объектов: object и shape.
    - Object — объект, котороый неразрывно связан со своим собственным материалом, например: небо, cubemap, текстурированный полигон.
    - Shape — объект, которому можно задать любой имеющийся материал, например, это объекты: сфера, цилиндр, полигон, треугольник, портал, контур (состоит из сфер и цилиндров).
  - Полиморфизм камер — можно рендерить картинку совершенно различными камерами, которые наследуются от абстрактного класса базовой камеры и реализуют виртуальные методы. Таким образом уже реализованы камеры: перспективная, ортогональная, 360. Так же можно реализовать другие камеры, например: проекцию Панини, камера для создания cubemap.
  - Полиморфизм материалов — любому объекту можно задать любой материал на его поверхности, например реализованы такие материалы, как: рассеивающий, отражающий, стекло, освещение. Аналогично для реализации материала надо наследоваться от базового класса материала и реализовать метод обработки луча материалом.
- Точечные источники света поддерживаются на уровне рендерера.
- Рендеринг порталов. Портал является shape, потому что можно задать какой материал будет на двух его задних частях. Он сам задается как полигон и две системы координат.

    - Поддерживается **телепортирование освещения** от точечных источников света через порталы. При этой телепортации учитывается, что некоторая часть света может не пройти через портал.
- **Считывание произвольной сцены из многоугольников, с анимацией, из `json` файла.**
- **Поддержка текстур.** Реализуется эта поддержка в классе текстурированного полигона.
- Рендеринг произвольных полигонов.
- Для path tracing камерой поддерживается симуляция прохождения луча через диафрагму, поэтому можно получить эффект **depth of field**: когда камера на чем-то сфокусирована, а остальная часть мира размыта.
- Рендеринг происходит в **многопоточном режиме**.
- Имеется возможность одновременно с обычным изображением рендерить изображение глубины.
- Имеется возможность сохранять отрендеренные изображения в png.
- При рендеринге пиксели выбираются случайным образом, чтобы максимально точно предсказывать итоговое время рендеринга, которое выводится на экран.
- Возможность задавать количество семплов для одного пикселя. В данной работе обычно выбирается 16 семплов для ray tracing, и 400 для path tracing.

4 Пример реализованных функций

4.1 Различие Ray tracing от Path tracing



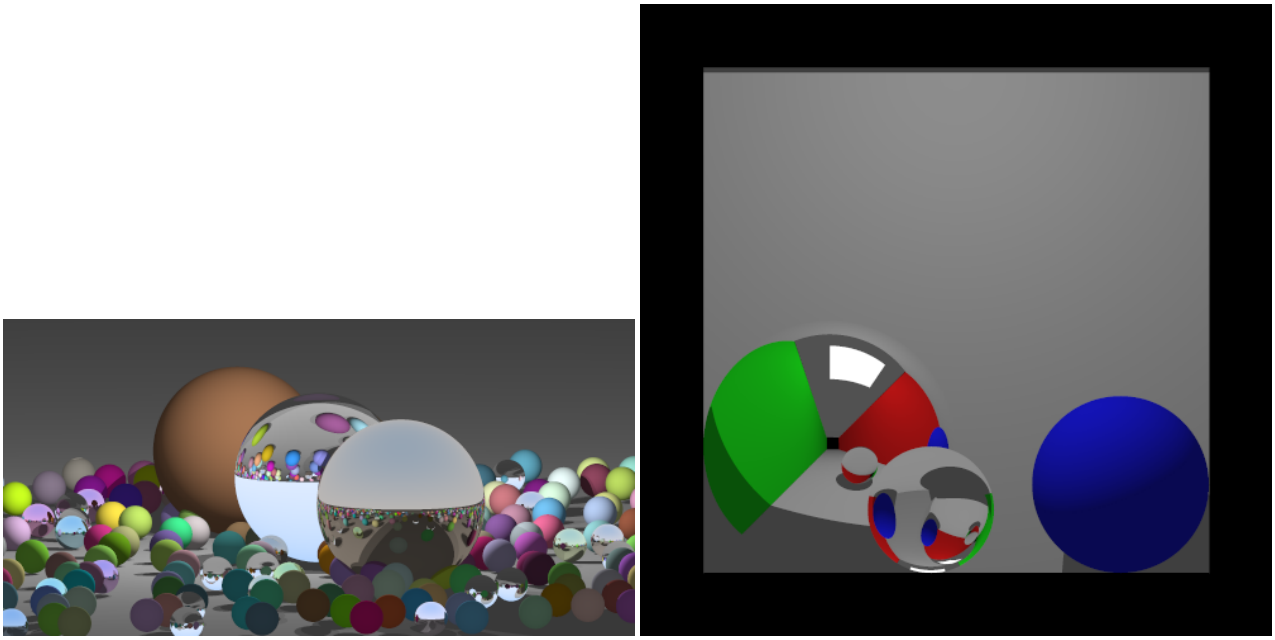


Слева — Ray Tracing, справа — Path Tracing. На первой сцене присутствуют точечные источники света, а на второй точечный источник света присутствует только для ray tracing, поэтому картинка с path tracing выглядит такой темной. потому что весь свет там получится от светящегося полигона сверху, вероятность попадания в который невелика.

Слева освещение получается только от фонового освещения и точечных источников света. Справа же освещение получается от трассировки вторичных лучей, которые идут в случайном направлении от рассеивающего материала. Благодаря этому получается очень фотореалистичная картинка, а так же такие эффекты, как: глубина резкости — размытие ближних и дальних объектов, не попавших в фокус; каустика — явление попадания большого количества света в одно место, её можно наблюдать под стеклянной сферой на картинке с комнатой; освещение от зелёной и красной стены на полу рядом со стеклянной сферой — тоже эффект от вторичного трейсинга лучей; освещение потолка зелёным цветом — от зелёной стены.

4.2 Различные виды камер

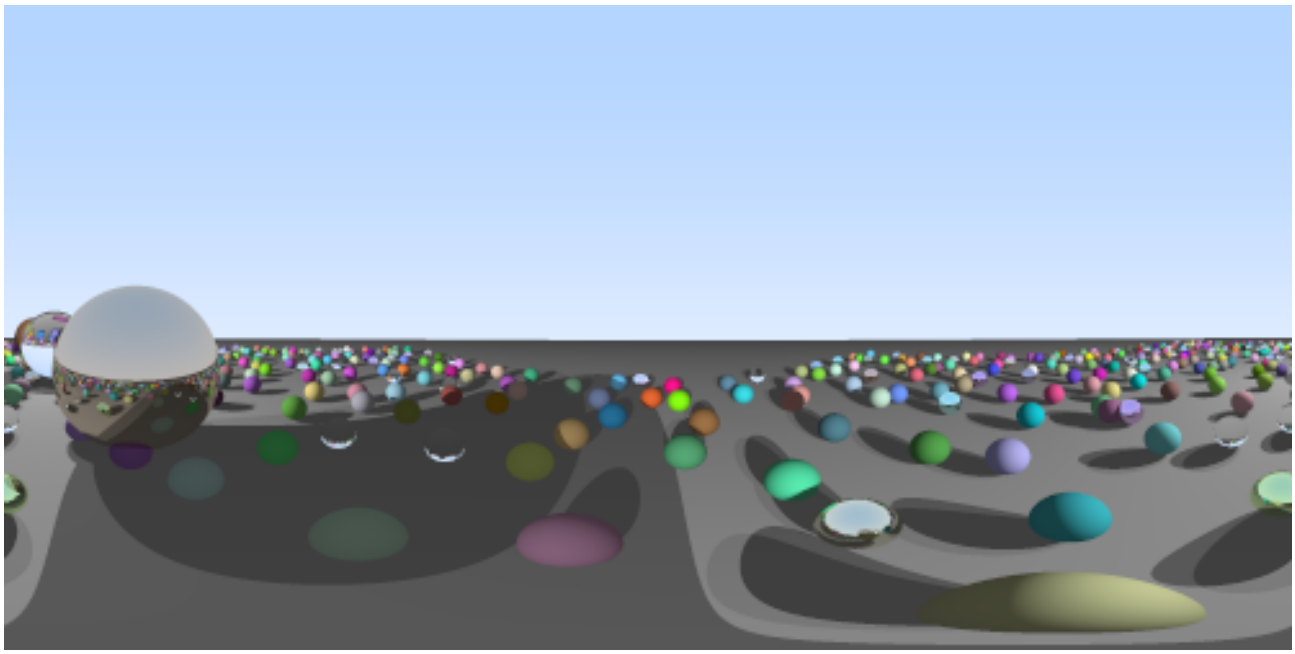
4.2.1 Ортогональная

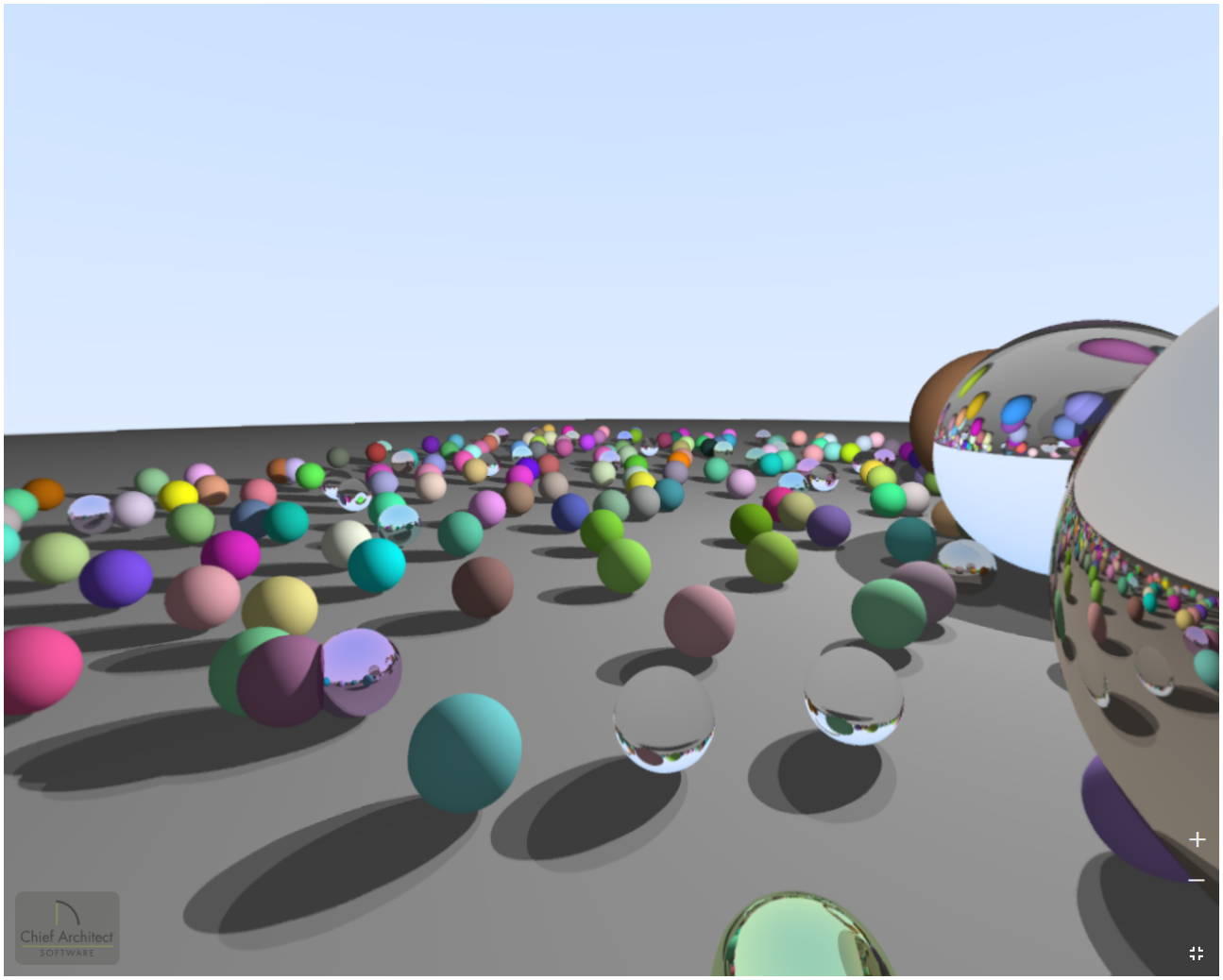


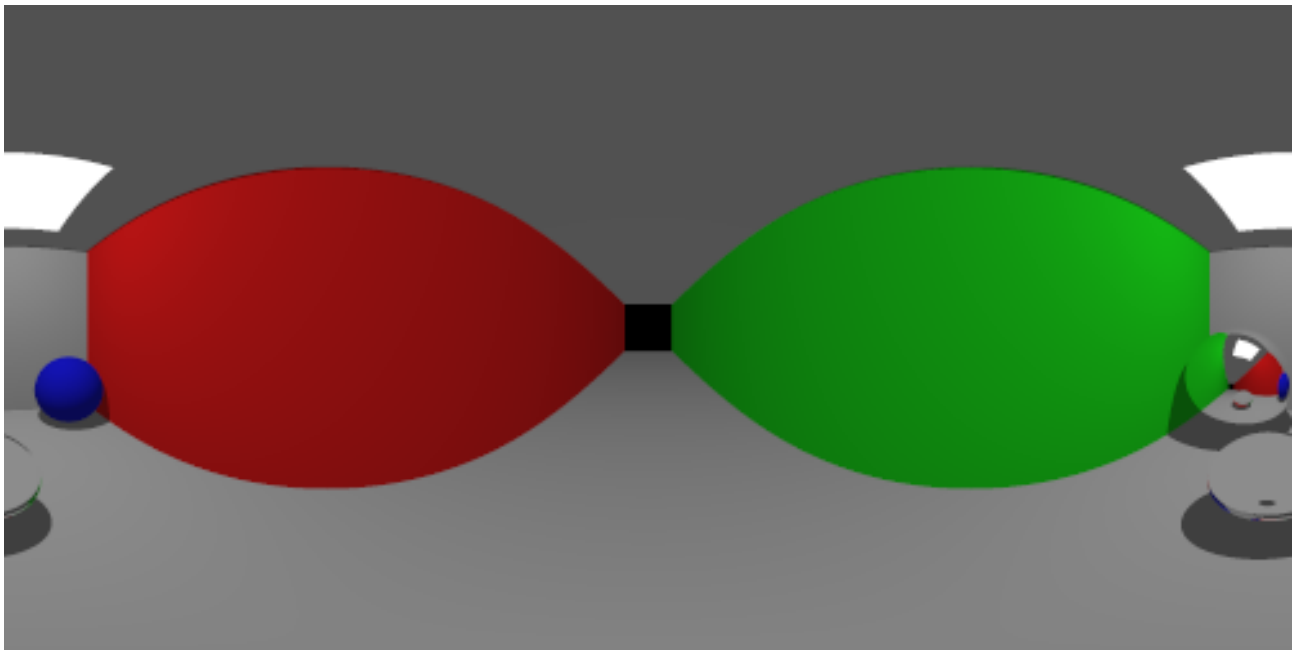
4.2.2 Камера на 360 градусов

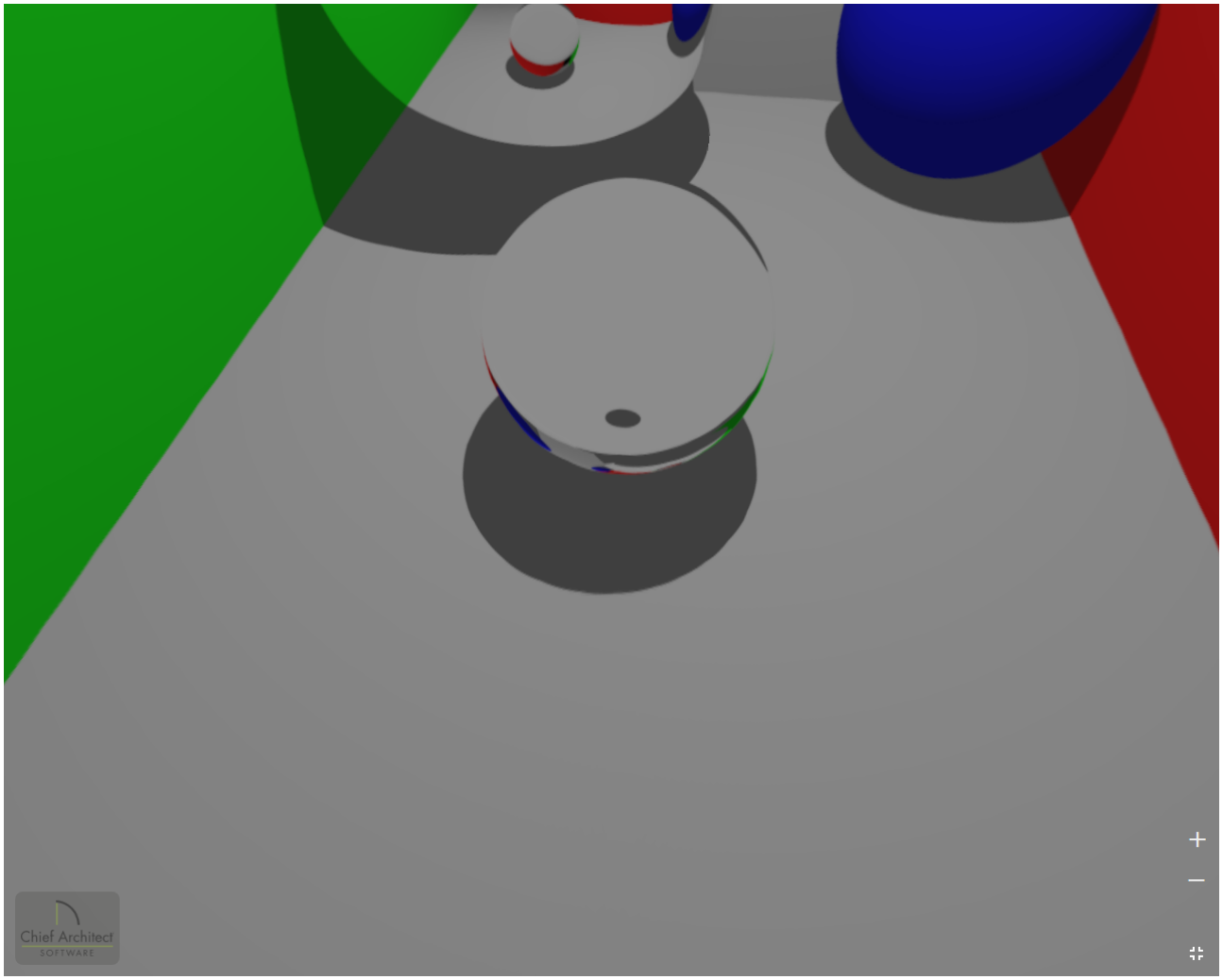
Эта камера создает такое изображение, которое запечатляет всю обстановку вокруг. Далее, по этому изображению можно получить сколько угодно изображений перспективной проекции, которые смотрят с разных углов при помощи различных программ для просмотра такого рода изображений.

Изображения в 360 градусов были просмотрены с помощью www.chiefarchitect.com.




Chief Architect
SOFTWARE

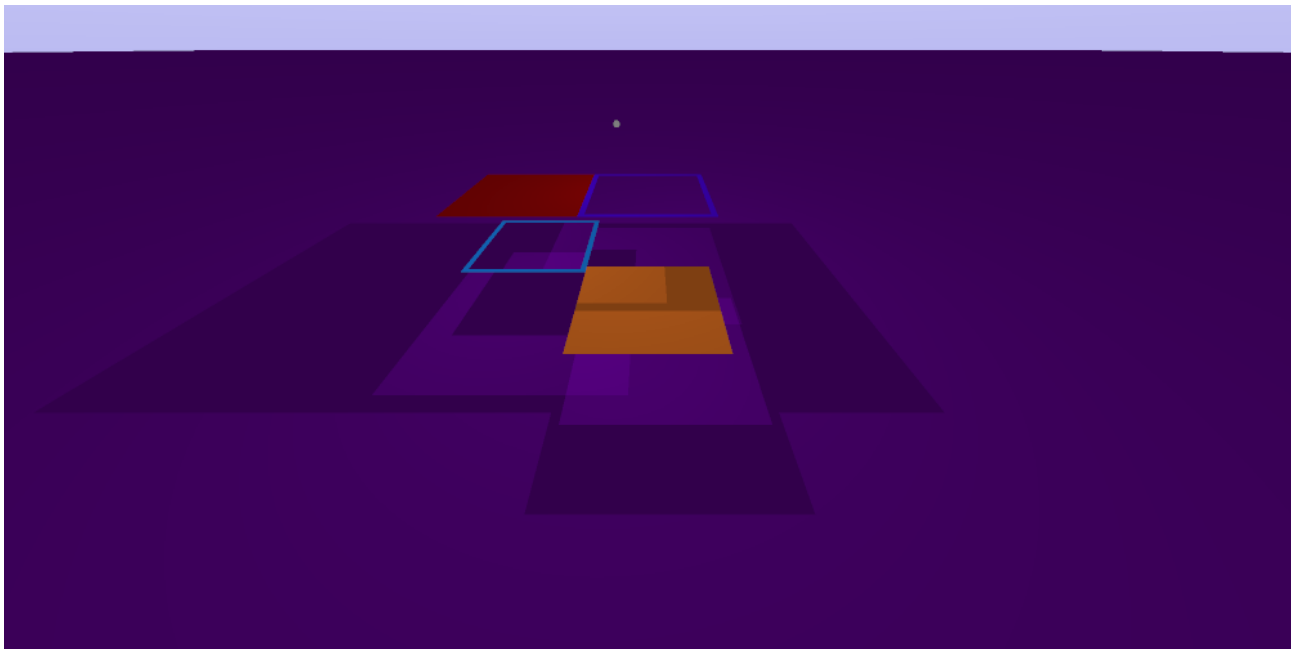


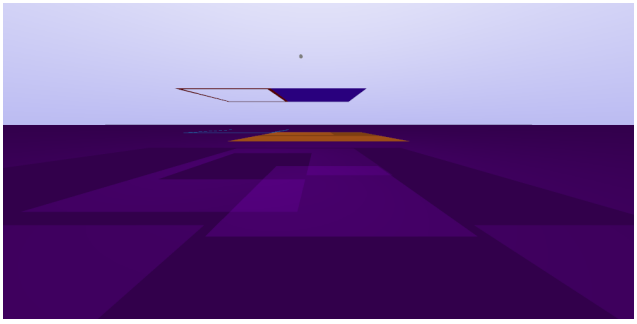


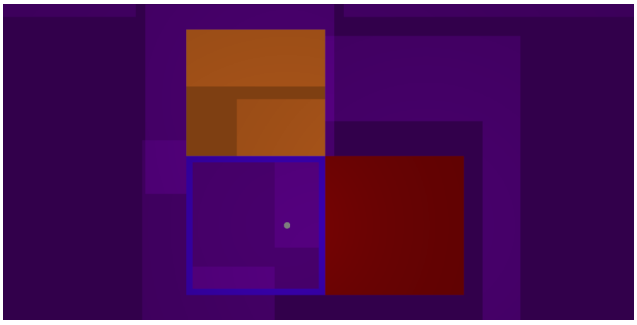
4.3 Телепортация освещения от точечного источника света через портал

Маленькой сферой показано положение точечного источника света. Темно-синий портал связан с красным, а голубой с оранжевым. Свет проходит через все порталы и освещает пол снизу. Так же некоторые части освещения от этого же источника света накладываются друг на

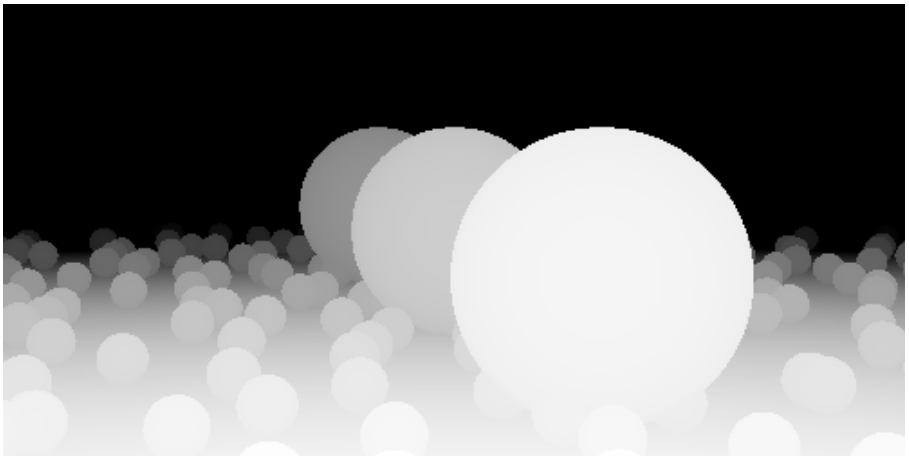
друга, хотя источник света один. Эффект примерно такой же, как и от зеркал, но работает с порталами.



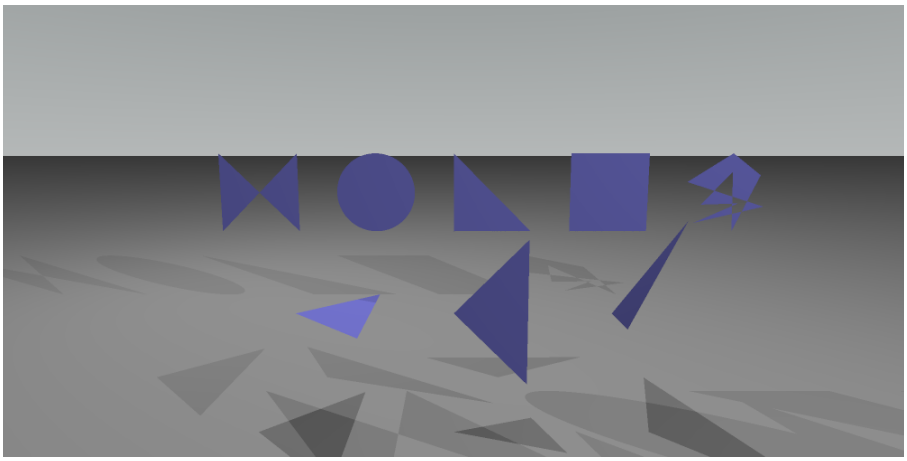




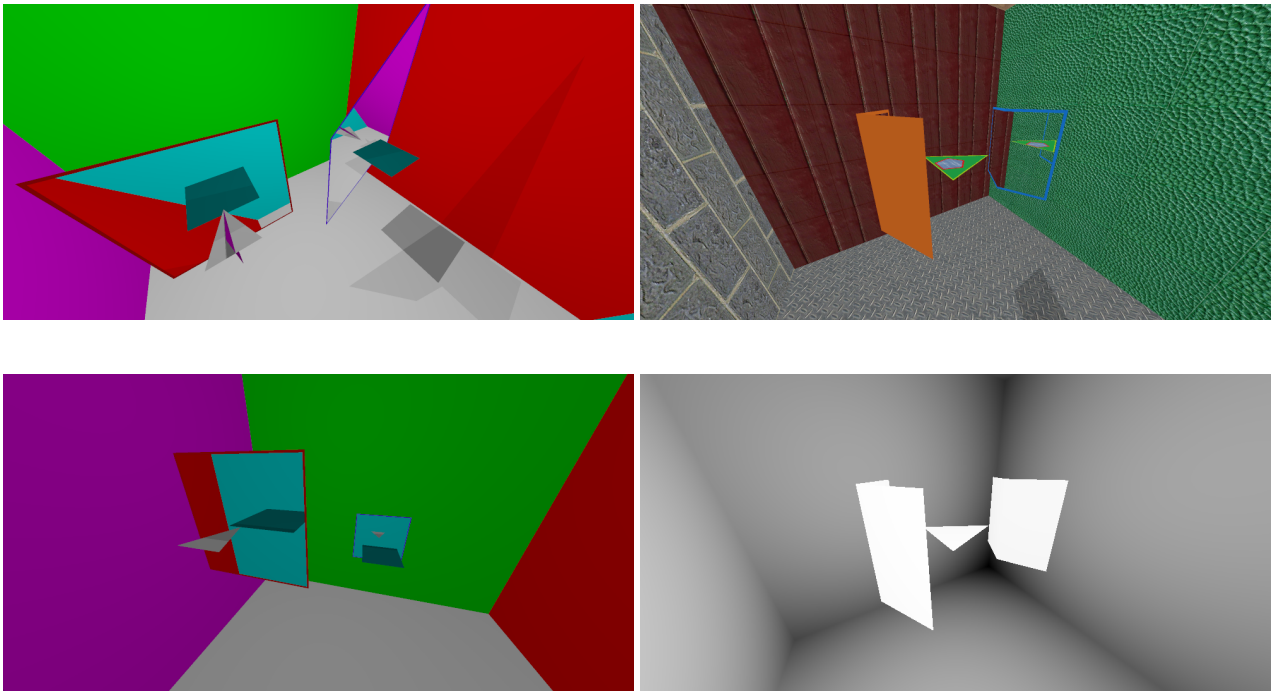
4.4 Изображение глубины каждого пикселя



4.5 Рендеринг произвольных полигонов

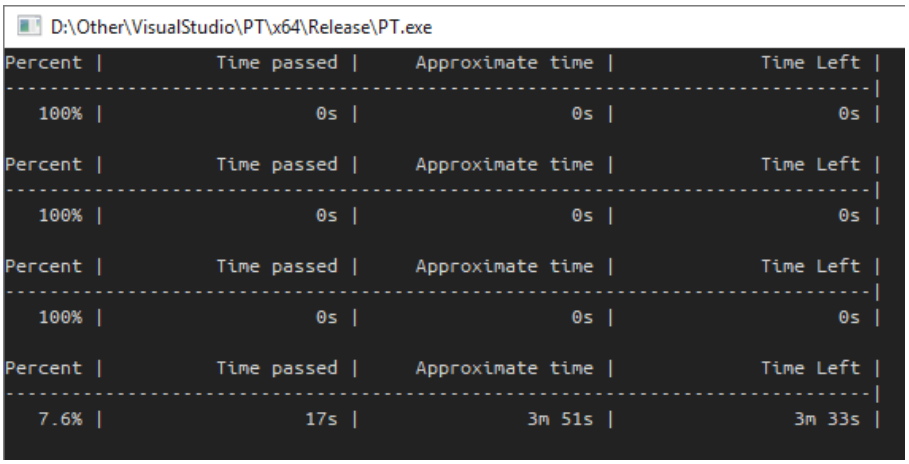


4.6 Рендеринг сцен из предыдущей работы, заданных в json



4.7 Пример предсказания времени рендеринга

Так как рендеринг изображения при помощи path tracing может занимать часы, иногда бывает необходимо знать точное время рендеринга. Специально для этого каждые n отрендеренных пикселей собирается информация о времени рендеринга, находится процент завершенной работы и предсказывается время всего рендеринга. Так же для увеличения точности, все пиксели для рендеринга берутся в случайном порядке. Если же не делать этого случайного выбора пикселей, то у нас на изображении, у которого сверху небо, и соответственно, нет никаких вторичных лучей, будет очень быстро отрендерена верхняя часть, в то время, когда нижняя часть будет рендериться очень долго. Поэтому пиксели надо брать в случsайном порядке.



5 Код программы

Код находится в открытом доступе по адресу: path-tracing. Зависимости: space-objects — для вычислений с системами координат; stb — для сохранения результата в png; json, portals-opengl — для работы кода по считыванию сцены из json.

5.1 Файлы заголовков

FILE **basics.h**

```
#ifndef PT_BASICS_H
#define PT_BASICS_H

#include <memory>

#include <spob/spob.h>
#include <pt/poly.h>

namespace pt
{

    // Используем это сущности из spob
    using namespace spob;

    //-------------------------------------------------------------------------
    const double pi = 3.14159265358979323846264338327 9;
    double random(void);

    //-------------------------------------------------------------------------
    class Color
    {
    public:
        double r, g, b, a;
        Color() : r(0), g(0), b(0), a(0) {}
        Color(double r, double g, double b, double a = 1) : r(r*r), g(g*g), b(b*b), a(a) {}

        /** Делит цвет на некоторое число для взятия среднего арифметического среди нескольких
        ↪  цветов. Среднее арифметическое так же берется и для альфа канала. */
        Color operator/=(double k);

        /** Умножает два цвета, только в пространстве r, g, b. Если один из цветов Transparent, то
        ↪  возвращает текущий цвет.

        ИЛИ. Считает, что оба цвета полупрозрачные и считает как если бы текущий накладывался поверх
↪  a. */
        Color operator*(const Color& a) const;

        Color operator*(double a) const;
        Color operator+=(const Color& a);

```

```
        /** Получает квадратный корень из цвета. При этом квадратный корень из альфа-канала не
        ↪  берется. */
        Color sqrt(void);
    };

    const Color transparent(0, 0, 0, 0);

    Color overlay(const Color& upper, const Color& lower);

    //------------------------------------------------------------------------
    void reflect(vec3& ray, const vec3& normal);
    bool refract(vec3& ray, const vec3& normal, double refractiveIndex);
    vec3 randomSphere(void);

};

#endif
```

FILE **camera.h**

```
#ifndef PT_CAMERA_H
#define PT_CAMERA_H

#include <pt/basics.h>
#include <pt/object.h>

namespace pt
{

    vec3 getRotatedVector(const vec3& pos, double r, double alpha, double beta);

    //------------------------------------------------------------------------
    class Camera
    {
    public:
        Camera(vec3 pos) : pos(pos) {}

        virtual ~Camera() {}
        virtual Ray getRay(double x, double y, bool isDiffuse) const = 0;

        vec3 pos;
    };

    //------------------------------------------------------------------------
    class PerspectiveCamera : public Camera
    {
    public:
        PerspectiveCamera(double focal,
                          double viewAngle,
                          double aperture,
                          vec3 pos,
                          double width,
                          double height);

        void assign(double focal1,
                    double viewAngle1,
                    double aperture1,
                    vec3 pos1,
                    double width1,
                    double height1);

        Ray getRay(double x, double y, bool isDiffuse) const;

        void lookAt(const vec3& towards);

        double focal;
        double aperture;
        double viewAngle;
        vec3 i, j, k;
        double h;
        double width, height;
    };

};

```

```
#endif
```

FILE **image.h**

```
#ifndef PT_IMAGE_H
#define PT_IMAGE_H

#include <memory>
#include <string>
#include <pt/basics.h>

namespace pt
{

    class Image
    {
    public:
        Image(int width, int height);
        Image(const Image& img);
        Image(const Image* img);
        ~Image();

        void resize(int width, int height);

        /** Берет из цветов квадратный корень, потому что монитор представляет цвет пикселя не
        ↪  линейно возрастающий, а нелинейно, и так как этот рисунок - результат рендеринга, то
        ↪  здесь так же надо представлять цвета мира, чтобы они корректно отображались на экране. */
        void colorCorrection(void);

        void clear(void);

        int getWidth() const;
        int getHeight() const;
        Color& operator()(int x, int y);
        const Color& operator()(int x, int y) const;
    private:
        Color*    m_pix;
        int     m_width;
        int       m_height;
    };

    typedef std::shared_ptr<Image> Image_ptr;

    //-------------------------------------------------------------------------
    void saveAsDoubleImg(const Image& img, const std::string& name);
    void loadAsDoubleImg(Image& img, const std::string& name);

    void toGrayScaleDoubleImg(Image& img, double overrideMax = -1);

};

#endif
```

FILE **object.h**

```
#ifndef PT_OBJECT_H
#define PT_OBJECT_H

#include <pt/basics.h>

namespace pt
{

    struct Ray;
    struct Intersection;
    class Object;
    class Material;
    class Object;

    typedef std::shared_ptr<Object> Object_ptr;
    typedef std::shared_ptr<Material> Material_ptr;

    //-------------------------------------------------------------------------
    struct Ray
    {
```

```
21          vec3 pos;
22          vec3 dir;
23      };
24
25      //-----------------------------------------------------------------------
26      struct Intersection
27      {
28          double t;
29          vec3 pos;
30          vec3 normal;
31          struct Data {
32              int type;
33              int integer;
34              vec2 vector;
35              bool boolean;
36          } data;
37      };
38
39      enum ScatterType
40      {
41          SCATTER_END, // Возвращается, если луч поглотился, или достигнут источник света. Если луч
            ↪  поглотился, то объект обязан записать в clrAbsorbtion нулевой цвет. Если достигнут
            ↪  источник света, то туда записывается цвет света.
42          SCATTER_RAYTRACING_END, // Возвращается, если рейтрейсинг может остановиться на этом. Такое
            ↪  может возвращаться например для: обычного материала, подповерхностного рассеивания.
43          SCATTER_NEXT // Возвращается, если луч может идти дальше, например для: искривителей
            ↪  лучей(черные дыры, червоточины), прозрачные объекты, отражающие объекты.
44      };
45
46      //-----------------------------------------------------------------------
47      class Object
48      {
49      public:
50          virtual ~Object() {}
51          virtual bool intersect(const Ray& ray,
52                                 Intersection& inter,
53                                 double tMin,
54                                 double tMax) const = 0;
55
56          /** Получает текущий луч, его пересечение с фигурой. Возвращает поглощенный цвет и
            ↪  направление отраженного луча. Функция возвращает true, если луч может идти дальше, false,
            ↪  если луч поглотился. Тогда, если он поглотился, в цвет поглощения обязано быть записано
            ↪  0, если этот объект не является источником света. Если является источником света, то в
            ↪  цвет поголщения должен быть записан цвет свечения. */
57          virtual ScatterType scatter(const Ray& ray,
58                                      const Intersection& inter,
59                                      Color& clrAbsorbtion,
60                                      Ray& scattered,
61                                      /** Показывает насколько сильно данный луч может рассеиваться в
                                        ↪  этом положении. Используется исключительно рендерерами.
                                        ↪  Объект должен каждый раз задавать этот параметр. */
62                                      double& diffusion) const = 0;
63      };
64
65      //-----------------------------------------------------------------------
66      class Material
67      {
68      public:
69          virtual ~Material() {}
70          virtual ScatterType scatter(const Ray& ray,
71                                      const Intersection& inter,
72                                      Color& clrAbsorbtion,
73                                      Ray& scattered,
74                                      double& diffusion) const = 0;
75      };
76
77      //-----------------------------------------------------------------------
78      class Shape : public Object
79      {
80      public:
81          Shape(Material_ptr material) : material(material) {}
82
83          virtual bool intersect(const Ray& ray,
84                                 Intersection& inter,
85                                 double tMin,
```

```
                                double tMax) const = 0;
        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const final {
            return material->scatter(ray, inter, clrAbsorbtion, scattered, diffusion);
        }

        Material_ptr material;
    };

};

#endif
```

FILE **poly.h**

```
#pragma once

#include <vector>
#include <spob/spob.h>

bool pointInPolygon(const std::vector<spob::vec2>& poly, spob::vec2 p);
```

FILE **pt.h**

```
#ifndef PT_PT_H
#define PT_PT_H

#include <pt/basics.h>
#include <pt/camera.h>
#include <pt/image.h>
#include <pt/object.h>
#include <pt/renderer.h>
#include <pt/transformation.h>

#endif
```

FILE **renderer.h**

```
#ifndef PT_RENDERER1_H
#define PT_RENDERER1_H

#include <vector>

#include <pt/basics.h>
#include <pt/camera.h>
#include <pt/image.h>
#include <pt/object.h>
#include <pt/shape/portals.h>

namespace pt
{

    //-----------------------------------------------------------------------------
    /** Точечный источник света, характеризуется положением и цветом. Так как все цвета
    ↪  нормализуются, то от цвета берется квадратный корень, чтобы освещение на самом деле вело себя
    ↪  как освещение. */
    struct PointLight
    {
        PointLight(vec3 pos, Color clr) : pos(pos), clr(clr.sqrt()) {}

        vec3 pos;
        Color clr;
    };

    //-----------------------------------------------------------------------------
    /** Абстрактный класс рендерера. */
    class Renderer1
    {
    public:
        Renderer1() : isDrawDepth(false) {}
        virtual ~Renderer1() {}

```

```
        void assign(Camera* camera1, Object* scene1, Image* img1) {
            camera = camera1;
            scene = scene1;
            img = img1;
            isDrawDepth = false;
        }

        void assign(Camera* camera1, Object* scene1, Image* img1, Image* dImg1) {
            camera = camera1;
            scene = scene1;
            img = img1;
            dImg = dImg1;
            isDrawDepth = true;
        }

        virtual void render(void) = 0;

        struct Frag
        {
            Frag(Color color, double depth) : color(color), depth(depth) {}
            Color color;
            double depth;
        };

        virtual Frag computePixel(int x, int y) const = 0;
        virtual Frag computeColor(Ray ray) const = 0;
    protected:
        Camera*    camera;
        Object*    scene;
        Image*    img;
        Image*    dImg;
        bool    isDrawDepth;

        virtual void onStartRender(void) {}
        virtual void onEveryLine(double percent) const {}
        virtual void onEndRendering(void) const {}
    };

    //---------------------------------------------------------------------------
    /** Стандартный класс рендеринга, на основе которого строятся ray- и path-tracing. Имеет в себе
    ↪  не только простой рендеринг, но еще и фичу: точечные источники освещения, которые
    ↪  поддерживают не только полупрозрачные объекты, но еще и порталы. Вообще освещение можно было
    ↪  бы моделировать при помощи сфер с материалом light, но это будет слишком долго. */
    class StandardRenderer : public Renderer1
    {
    public:
        /** Инициализация рендеринга. Для каждой новой картинки необходимо заново инициализировать
        ↪  рендерер.
            Для ray-tracing: isDiffuse = false, isBreakOnMaterial = true.
            Для path-tracing: isDiffuse = true, isBreakOnMaterial = false. */
        StandardRenderer(int threads,
                         int maxDepth,
                         double tMax,
                         bool isDiffuse,
                         bool isBreakOnMaterial,
                         bool isWriteText);

        ~StandardRenderer();

        /** Рендерит в заданную картинку с заданной камерой, сценой и параметрами. Порядок рендеринга
        ↪  - случайные пиксели, это дает возможность хорошо предсказывать время рендеринга. */
        void render(void);

        /** Добавляет во внутреннее хранилище портал для последующей обработки. */
        void addPortal(Portals_ptr portal);

        /** Удаляет все порталы из обработки рендером. */
        void clearPortals(void);

        Frag computePixel(int x, int y) const;
        Frag computeColor(Ray ray) const;

        /** Устанавливает фоновое освещение. */
        void setAmbientLight(Color clr);

```

```
103         /** Массив точечных источников освещения. Пользователь сам их задает, далее это учитывается
            ↪  при рендеринге. */
104         std::vector<PointLight>    luminaries;
105     protected:
106         std::vector<Portals_ptr> portals;
107         std::vector<Portals_ptr> invertedPortals;
108 
109         int threads;
110         int maxDepth;
111         double tMax;
112         bool isDiffuse;
113         bool isBreakOnMaterial;
114         bool isWriteText;
115         Color ambient;
116 
117         double time;
118 
119         /** Эти функции выводят информацию в консоль. */
120         void onStartRender(void);
121         void onEveryLine(double percent) const;
122         void onEndRendering(void) const;
123 
124         /** Считает все возможное освещение от точечных источников освещения в данной позиции сцены,
            ↪  при этом учитывается наличие порталов, а так же то, что объекты могут быть полупрозрачны.
            ↪  */
125         Color computeLight(vec3 pos, vec3 normal,
126                            std::vector<std::pair<Portals_ptr, vec3> >& teleportation,
127                            int depth) const;
128     };
129 
130     //-------------------------------------------------------------------------
131     /** Рендерит так называемый ray-tracing. Особенности:
132             Достаточно одного луча на пиксель.
133             Малографонистое изображение.
134             Поддержка точечных источников освещения(единственный источник освещения для данного
    ↪  рендеринга).
135                 Соответственно тени могут быть только точными.
136                 Учитываются полупрозрачные объекты.
137                 Учитываются порталы.
138             Не поддерживает depth of field.
139             Не поддерживает рассеянные тени и рассеянное освещение(например от неба, от источников
    ↪  освещения, имеющих форму).
140      */
141     class RayTracing : public StandardRenderer
142     {
143     public:
144         RayTracing(int aliasing = 1,
145                    int threads = 1,
146                    bool isWriteText = true,
147                    int maxDepth = 30,
148                    double tMax = 100000);
149 
150     protected:
151         int antialiasing;
152         Frag computePixel(int x, int y) const;
153     };
154 
155     //-------------------------------------------------------------------------
156     /** Рендерит так называемый path-tracing. Особенности:
157             Недостаточно одного луча на пиксель.
158                 Чем больше лучей, тем лучше.
159                 Нормальное изображение - 400 лучей.
160                 Идеальное изображение - 10 000 лучей.
161             Графонистое изображение.
162             Поддержка точечных источников освещения(единственный источник освещения для данного
    ↪  рендеринга).
163                 Соответственно тени могут быть только точными.
164                 Учитываются полупрозрачные объекты.
165                 Учитываются порталы.
166             Поддержка depth of field.
167             Поддержка рассеянных теней и рассеянного освещения(например от неба, от источников
    ↪  освещения, имеющих форму).
168      */
169     class PathTracing : public StandardRenderer
170     {
```

```
    public:
        PathTracing(int samples = 400,
                    int threads = 1,
                    bool isWriteText = true,
                    int maxDepth = 30,
                    double tMax = 100000);

    protected:
        int samples;
        Frag computePixel(int x, int y) const;
    };

};

#endif
```

FILE **pt2easybmp.h**

```
#ifndef PT_PT2EASYBMP_H
#define PT_PT2EASYBMP_H

#include <string>
#include <pt/image.h>

namespace pt
{

    void saveAsPng(Image& img, std::string name);
    void loadAsPng(Image& img, std::string name);

};

#endif
```

5.1.1 camera/

FILE **360.h**

```
#ifndef PT_360_H
#define PT_360_H

#include <pt/camera.h>

namespace pt
{

    class Camera360 : public Camera
    {
    public:
        Camera360(vec3 pos, double resolution);

        Ray getRay(double x, double y, bool isDiffuse) const;

        double resolution;
    };

};

#endif
```

FILE **orthogonal.h**

```
#ifndef PT_ORTHOGONAL_H
#define PT_ORTHOGONAL_H

#include <pt/camera.h>

namespace pt
{

    class Orthogonal : public Camera
    {
```

```
    public:
        Orthogonal(vec3 pos, double scale, double width, double height);

        Ray getRay(double x, double y, bool isDiffuse) const;

        /** Направляет ортогональную камеру в указанную точку. */
        void lookTowards(vec3 toward);

        double scale;
        double width;
        double height;
    private:
        vec3 i, j, k;
    };

};

#endif
```

5.1.2 material/

FILE **light.h**

```
#ifndef PT_LIGHT_H
#define PT_LIGHT_H

#include <pt/object.h>

namespace pt
{

    class Light : public Material
    {
    public:
        Light(Color clr);

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        Color clr;
    };

    inline Material_ptr makeLight(Color clr) { return Material_ptr(new Light(clr)); }

};

#endif
```

FILE **reflect.h**

```
#ifndef PT_REFLECT_H
#define PT_REFLECT_H

#include <pt/object.h>

namespace pt
{

    class Reflect : public Material
    {
    public:
        Reflect(Color clr, double diffusion);

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        Color clr;
```

```
        double diffuse;
    };

    inline Material_ptr makeReflect(Color clr, double diffusion) { return Material_ptr(new
    ↪  Reflect(clr, diffusion)); }

};

#endif
```

FILE **refract.h**

```
#ifndef PT_REFRACT_H
#define PT_REFRACT_H

#include <pt/object.h>

namespace pt
{

    class Refract : public Material
    {
    public:
        Refract(double refractiveIndex, double diffusion);

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        double refractiveIndex;
        double diffuse;
    };

    inline Material_ptr makeRefract(double refractiveIndex, double diffusion) { return
    ↪  Material_ptr(new Refract(refractiveIndex, diffusion)); }

};

#endif
```

FILE **scatter.h**

```
#ifndef PT_SCATTER_H
#define PT_SCATTER_H

#include <pt/object.h>

namespace pt
{

    class Scatter : public Material
    {
    public:
        Scatter(Color clr, double k_coef = 1, double s_coef = 0);

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        Color clr;
        double s_coef;
        double k_coef;
    };

    inline Material_ptr makeScatter(Color clr, double k_coef = 1, double s_coef = 0) { return
    ↪  Material_ptr(new Scatter(clr, k_coef,  s_coef)); }

};

#endif
```

5.1.3 object/

FILE **cubemap.h**

```
#ifndef PT_CUBEMAP_H
#define PT_CUBEMAP_H

#include <string>
#include <EasyBMP.h>
#include <pt/object.h>

namespace pt
{

    class CubeMap : public Object
    {
    public:
        CubeMap(std::string bmp);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;
    private:
        BMP m_image;
        int m_size;
    };

    inline Object_ptr makeCubeMap(std::string bmp) { return Object_ptr(new CubeMap(bmp)); }

};

#endif
```

FILE **scene.h**

```
#ifndef PT_SCENE_H
#define PT_SCENE_H

#include <vector>
#include <pt/object.h>
#include <prtl_vis/scene_reader.h>

namespace pt
{

    class Scene : public Object
    {
    public:
        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        void add(Object_ptr obj);

        std::vector<Object_ptr> array;
    };

    Scene loadFrame(const scene::Frame& frame);

};

#endif
```

FILE **sky.h**

```
#ifndef PT_SKY_H
#define PT_SKY_H

#include <pt/object.h>

namespace pt
{

    class Sky : public Object
    {
    public:
        Sky(Color clr1, Color clr2);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;

        Color clr1;
        Color clr2;
    };

    inline Object_ptr makeSky(Color clr1, Color clr2) { return Object_ptr(new Sky(clr1, clr2)); }

};

#endif
```

FILE **texture_polygon.h**

```
#ifndef PT_TEXTURE_POLYGON_H
#define PT_TEXTURE_POLYGON_H

#include <pt/object.h>
#include <pt/image.h>
#include <pt/shape/polygon.h>
#include <pt/material/scatter.h>

namespace pt
{

    class TexturePolygon : public Object
    {
    public:
        TexturePolygon(const std::vector<vec2>& polygon, crd3 coords, Image_ptr img, const space2&
        ↪  tr, double k_coef = 1, double s_coef = 0);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;

        ScatterType scatter(const Ray& ray,
                            const Intersection& inter,
                            Color& clrAbsorbtion,
                            Ray& scattered,
                            double& diffusion) const;
    private:
        Image_ptr img;

        std::vector<vec2> array;
        crd3 coords;

        double d;
        vec3 normal;
        vec2 min, max;

        double k_coef, s_coef;

```

```
        space2 tr;
    };

    inline Object_ptr makeTexturePolygon(const std::vector<vec2>& polygon, crd3 coords, Image_ptr
    ↪  img, const space2& tr, double k_coef = 1, double s_coef = 0) { return Object_ptr(new
    ↪  TexturePolygon(polygon, coords, img, tr, k_coef, s_coef)); }

};

#endif
```

5.1.4 shape/

FILE **contour.h**

```
#ifndef PT_CONTOUR_H
#define PT_CONTOUR_H

#include <vector>
#include <pt/object.h>
#include <pt/shape/sphere.h>
#include <pt/shape/cylinder.h>
#include <pt/object/scene.h>
#include <pt/transformation.h>

namespace pt
{

    class Contour : public Shape
    {
    public:
        Contour(std::vector<vec3> array, double thick, bool isClosed, Material_ptr material) :
        ↪  Shape(material) {
            assign(array, thick, isClosed);
        }

        void assign(const std::vector<vec3>& array, double thick, bool isClosed);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
    private:
        Scene                   m_scene;
    };

    inline Object_ptr makeContour(std::vector<vec3> array, double thick, bool isClosed, Material_ptr
    ↪  material) { return Object_ptr(new Contour(array, thick, isClosed, material)); }

};

#endif
```

FILE **cylinder.h**

```
#ifndef PT_CYLINDER_H
#define PT_CYLINDER_H

#include <pt/object.h>

namespace pt
{

    class Cylinder : public Shape
    {
    public:
        Cylinder(vec3 a, vec3 b, double r, Material_ptr material) : Shape(material), A(a), B(b), r(r)
        ↪  {}

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
```

```
    private:
        vec3 A, B;
        double r;
    };

    inline Object_ptr makeCylinder(vec3 a, vec3 b, double r, Material_ptr material) { return
    ↪  Object_ptr(new Cylinder(a, b, r, material)); }

};

#endif
```

FILE **polygon.h**

```
#ifndef PT_POLYGON_H
#define PT_POLYGON_H

#include <vector>
#include <pt/object.h>

namespace pt
{

    class Polygon : public Shape
    {
    public:
        Polygon(const std::vector<vec2>& polygon, crd3 coords, Material_ptr material);

        void assign(const std::vector<vec2>& polygon, crd3 coords, Material_ptr material);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
    private:
        std::vector<vec2> array;
        crd3 coords;

        double d;
        vec3 normal;
        vec2 min, max;
    };

    inline Object_ptr makePolygon(const std::vector<vec2>& polygon, crd3 coords, Material_ptr
    ↪  material) { return Object_ptr(new Polygon(polygon, coords, material)); }

};

#endif
```

FILE **portals.h**

```
#ifndef PT_PORTALS_H
#define PT_PORTALS_H

#include <pt/object.h>
#include <pt/shape/polygon.h>

namespace pt
{

    //-------------------------------------------------------------------------
    class Portals : public Object
    {
    public:
        Portals(crd3 c1, crd3 c2, std::vector<vec2> poly, Material_ptr first, Material_ptr second);
        ~Portals();

        void assign(crd3 c1, crd3 c2, std::vector<vec2> poly, Material_ptr first, Material_ptr
        ↪  second);

        bool intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const;
```

```
23
24          ScatterType scatter(const Ray& ray,
25                              const Intersection& inter,
26                              Color& clrAbsorbtion,
27                              Ray& scattered,
28                              double& diffusion) const;
29          Polygon pg1;
30          Polygon pg2;
31          space3 p1;
32          space3 p2;
33          std::vector<vec2> poly;
34          Material_ptr first;
35          Material_ptr second;
36      };
37
38      typedef std::shared_ptr<Portals> Portals_ptr;
39
40      inline Portals_ptr makePortals(crd3 c1, crd3 c2, std::vector<vec2> poly, Material_ptr first,
        ↪  Material_ptr second) { return Portals_ptr(new Portals(c1, c2, poly, first, second)); }
41
42      //------------------------------------------------------------------------
43      Portals invert(Portals a);
44
45  };
46
47  #endif
```

FILE **sphere.h**

```
1   #ifndef PT_SPHERE_H
2   #define PT_SPHERE_H
3
4   #include <pt/object.h>
5
6   namespace pt
7   {
8
9       class Sphere : public Shape
10      {
11      public:
12          Sphere(vec3 a, double r, Material_ptr material) : Shape(material), A(a), r(r) {}
13
14          bool intersect(const Ray& ray,
15                         Intersection& inter,
16                         double tMin,
17                         double tMax) const;
18      private:
19          vec3 A;
20          double r;
21      };
22
23      inline Object_ptr makeSphere(vec3 a, double r, Material_ptr material) { return Object_ptr(new
        ↪  Sphere(a, r, material)); }
24
25  };
26
27  #endif
```

FILE **triangle.h**

```
1   #ifndef PT_TRIANGLE_H
2   #define PT_TRIANGLE_H
3
4   #include <pt/object.h>
5
6   namespace pt
7   {
8
9       class Triangle : public Shape
10      {
11      public:
12          Triangle(vec3 a, vec3 b, vec3 c, Material_ptr material);
13
14          bool intersect(const Ray& ray,
15                         Intersection& inter,
```

```
                        double tMin,
                        double tMax) const;

        vec3 a, b, c;
    private:
        vec3 normal;
        double d;
        double S;
        double ab, bc, ac;
    };

    inline Object_ptr makeTriangle(vec3 a, vec3 b, vec3 c, Material_ptr material) { return
    ↪  Object_ptr(new Triangle(a, b, c, material)); }

};

#endif
```

5.2 Исходные файлы

FILE **360.cpp**

```
#include <pt/camera/360.h>

namespace pt
{

//---------------------------------------------------------------------------
Camera360::Camera360(vec3 pos, double resolution) : Camera(pos), resolution(resolution) {}

//---------------------------------------------------------------------------
Ray Camera360::getRay(double x, double y, bool isDiffuse) const {
    x -= resolution;
    y -= resolution / 2.0;
    x *= pi/resolution;
    y *= pi/resolution;
    //x = -x;
    y = y + pi/2;
    Ray ray;
    ray.pos = pos;
    ray.dir = vec3(sin(y) * cos(x), sin(y) * sin(x), cos(y));
    ray.dir.normalize();
    return ray;
}

};
```

FILE **basics.cpp**

```
#include <random>
#include <mutex>
#include <math.h>
#include <pt/basics.h>

namespace pt
{

std::mt19937 generator;
std::uniform_real_distribution<double> distribution(0, 1);

//---------------------------------------------------------------------------
std::mutex m;
double random(void) {
    m.lock();
    double result = distribution(generator);
    m.unlock();
    return result;
}

//---------------------------------------------------------------------------
Color Color::operator/=(double k) {
    r /= k;
```

```
    g /= k;
    b /= k;
    a /= k;
    return *this;
}

//-----------------------------------------------------------------------------
Color overlay(const Color& upper, const Color& lower) {
    Color out;
    out.a = upper.a + lower.a * (1 - upper.a);
    out.r = (upper.r * upper.a + lower.r * lower.a * (1 - upper.a))/out.a;
    out.g = (upper.g * upper.a + lower.g * lower.a * (1 - upper.a))/out.a;
    out.b = (upper.b * upper.a + lower.b * lower.a * (1 - upper.a))/out.a;
    return out;
}

//-----------------------------------------------------------------------------
Color Color::operator*(const Color& a) const {
    Color clr = *this;
    clr.a = clr.a + a.a*(1 - clr.a);
    clr.r *= a.r;
    clr.g *= a.g;
    clr.b *= a.b;
    return clr;
}

//-----------------------------------------------------------------------------
Color Color::operator+=(const Color& clr) {
    a += clr.a;
    r += clr.r;
    g += clr.g;
    b += clr.b;
    return *this;
}

//-----------------------------------------------------------------------------
Color Color::operator*(double a) const {
    Color clr = *this;
    clr.r *= a;
    clr.g *= a;
    clr.b *= a;
    return clr;
}

//-----------------------------------------------------------------------------
Color Color::sqrt(void) {
    r = ::sqrt(r);
    g = ::sqrt(g);
    b = ::sqrt(b);
    return *this;
}

//-----------------------------------------------------------------------------
void reflect(vec3& ray, const vec3& normal) {
    ray = ray - normal * dot(ray, normal) / dot(normal, normal) * 2;
}

//-----------------------------------------------------------------------------
bool refract(vec3& ray, const vec3& normal, double r) {
    ray.normalize();
    double c =  -dot(normal, ray.normalize());
    double d = 1 - r*r*(1 - c*c);
    if (d > 0) {
        ray = ray*r + normal*(r*c - sqrt(d));
        return true;
    } else
        return false;
}

//-----------------------------------------------------------------------------
vec3 randomSphere(void) {
    /*double alpha = random() * 2 * pi;
    double beta = random() * 2 * pi;
    vec3 r;
    r.x = sin(alpha) * cos(beta);
```

```
    r.y = sin(alpha) * sin(beta);
    r.z = cos(beta);
    return r;*/
    // WTF?
    vec3 v;
    do {
        v = vec3(random(), random(), random()) * 2 - vec3(1);
    } while(v.length() > 1);
    return v;
}

};
```

FILE **camera.cpp**

```
#include <pt/camera.h>

namespace pt
{

//---------------------------------------------------------------------------
vec3 getRotatedVector(const vec3& pos, double r, double alpha, double beta) {
    vec3 pos1(vec3(sin(pt::pi/2 - beta) * cos(alpha), sin(pt::pi/2 - beta) * sin(alpha), cos(pt::pi/2
    ↪  - beta)));
    pos1 *= r;
    pos1 += pos;
    return pos1;
}

//---------------------------------------------------------------------------
PerspectiveCamera::PerspectiveCamera(double focal, double viewAngle, double aperture, vec3 pos,
↪  double width, double height) : Camera(pos) {
    assign(focal, viewAngle, aperture, pos, width, height);
}

void PerspectiveCamera::assign(double focal1, double viewAngle1, double aperture1, vec3 pos1, double
↪  width1, double height1) {
    viewAngle = viewAngle1;
    focal = focal1;
    aperture = aperture1;
    pos = pos1;
    width = width1;
    height = height1;
    h = 2 * tan(viewAngle/2.0);
}

//---------------------------------------------------------------------------
Ray PerspectiveCamera::getRay(double x, double y, bool isDiffuse) const {
    x = (x-width/2.0)/height*h;
    y = (height/2.0-y)/height*h;
    Ray ray;
    if (isDiffuse) {
        vec3 offset;
        double alpha = random() * 2 * pi;
        double r = random();
        offset.x = sin(alpha)*r * aperture;
        offset.y = cos(alpha)*r * aperture;
        offset = i * offset.x + j * offset.y;

        ray.dir -= offset / focal;
        ray.pos += offset;
    }
    ray.pos += pos;
    ray.dir += i * x + j * y + k;
    ray.dir.normalize();
    return ray;
}

//---------------------------------------------------------------------------
void PerspectiveCamera::lookAt(const vec3& towards) {
    k = towards - pos;
    k.normalize();
    i = cross(vec3(0, 0, 1), k).normalize();
    j = cross(k, i).normalize();
}
```

```

};
```

FILE **contour.cpp**

```
#include <pt/shape/contour.h>

namespace pt
{

//---------------------------------------------------------------------------
void Contour::assign(const std::vector<vec3>& array, double thick, bool isClosed) {
    for (int i = 0; i < array.size(); ++i) {
        m_scene.array.push_back(makeSphere(array[i], thick, nullptr));
        if ((i == array.size()-1 && isClosed) || (i != array.size()-1))
            m_scene.array.push_back(makeCylinder(array[i], array[i+1], thick, nullptr));
    }
}

//---------------------------------------------------------------------------
bool Contour::intersect(const Ray& ray,
                        Intersection& inter,
                        double tMin,
                        double tMax) const {
    return m_scene.intersect(ray, inter, tMin, tMax);
}

};
```

FILE **cubemap.cpp**

```
#include <pt/object/cubemap.h>

namespace pt
{

//---------------------------------------------------------------------------
void convert_xyz_to_cube_uv(double x, double y, double z, int *index, double *u, double *v) {
    // Thanks for Wikipedia: https://en.wikipedia.org/wiki/Cube_mapping
    double absX = fabs(x);
    double absY = fabs(y);
    double absZ = fabs(z);

    int isXPositive = x > 0 ? 1 : 0;
    int isYPositive = y > 0 ? 1 : 0;
    int isZPositive = z > 0 ? 1 : 0;

    double maxAxis, uc, vc;

    // POSITIVE X
    if (isXPositive && absX >= absY && absX >= absZ) {
        // u (0 to 1) goes from +z to -z
        // v (0 to 1) goes from -y to +y
        maxAxis = absX;
        uc = -z;
        vc = y;
        *index = 0;
    }
    // NEGATIVE X
    if (!isXPositive && absX >= absY && absX >= absZ) {
        // u (0 to 1) goes from -z to +z
        // v (0 to 1) goes from -y to +y
        maxAxis = absX;
        uc = z;
        vc = y;
        *index = 1;
    }
    // POSITIVE Y
    if (isYPositive && absY >= absX && absY >= absZ) {
        // u (0 to 1) goes from -x to +x
        // v (0 to 1) goes from +z to -z
        maxAxis = absY;
        uc = x;
        vc = -z;
        *index = 2;
```

```
    }
    // NEGATIVE Y
    if (!isYPositive && absY >= absX && absY >= absZ) {
        // u (0 to 1) goes from -x to +x
        // v (0 to 1) goes from -z to +z
        maxAxis = absY;
        uc = x;
        vc = z;
        *index = 3;
    }
    // POSITIVE Z
    if (isZPositive && absZ >= absX && absZ >= absY) {
        // u (0 to 1) goes from -x to +x
        // v (0 to 1) goes from -y to +y
        maxAxis = absZ;
        uc = x;
        vc = y;
        *index = 4;
    }
    // NEGATIVE Z
    if (!isZPositive && absZ >= absX && absZ >= absY) {
        // u (0 to 1) goes from +x to -x
        // v (0 to 1) goes from -y to +y
        maxAxis = absZ;
        uc = -x;
        vc = y;
        *index = 5;
    }

    // Convert range from -1 to 1 to 0 to 1
    *u = 0.5f * (uc / maxAxis + 1.0f);
    *v = 0.5f * (vc / maxAxis + 1.0f);
}

//---------------------------------------------------------------------------
CubeMap::CubeMap(std::string bmp) {
    std::wstring wstr(bmp.begin(), bmp.end());
    m_image.ReadFromFile(wstr.c_str());
    m_size = m_image.TellWidth()/4;
}

//---------------------------------------------------------------------------
bool CubeMap::intersect(const Ray& ray,
                    Intersection& inter,
                    double tMin,
                    double tMax) const {
    inter.t = tMax;
    inter.pos = ray.pos + ray.dir * inter.t;
    inter.normal = ray.dir;
    return true;
}

//---------------------------------------------------------------------------
ScatterType CubeMap::scatter(const Ray& ray,
                         const Intersection& inter,
                         Color& clrAbsorbtion,
                         Ray& scattered,
                         double& diffusion) const {
    vec3 dir = ray.dir;
    int index = 0;
    double u = 0, v = 0;

    convert_xyz_to_cube_uv(-dir.x, dir.z, dir.y, &index, &u, &v);
    v = 1 - v;
    u *= m_size;
    v *= m_size;

    switch (index) {
        case 0: {
            u += m_size * 2;
            v += m_size;
        } break;
        case 1: {
            v += m_size;
        } break;
```

```
        case 2: {
            u += m_size;
        } break;
        case 3: {
            u += m_size;
            v += m_size * 2;
        } break;
        case 4: {
            u += m_size;
            v += m_size;
        } break;
        case 5: {
            u += m_size * 3;
            v += m_size;
        } break;
    }

    auto pix = m_image.GetPixel(u, v);

    clrAbsorbtion.a = 1;
    clrAbsorbtion.r = pix.Red/255.0;
    clrAbsorbtion.g = pix.Green/255.0;
    clrAbsorbtion.b = pix.Blue/255.0;
    clrAbsorbtion.r *= clrAbsorbtion.r;
    clrAbsorbtion.g *= clrAbsorbtion.g;
    clrAbsorbtion.b *= clrAbsorbtion.b;
    diffusion = 0;

    return SCATTER_END;
}

};
```

FILE **cylinder.cpp**

```
#include <pt/shape/cylinder.h>

namespace pt
{

//-----------------------------------------------------------------------------
bool Cylinder::intersect(const Ray& ray,
                         Intersection& inter,
                         double tMin,
                         double tMax) const {
    vec3 V = ray.dir;
    vec3 P = ray.pos - A;
    vec3 D = B - A;

    double vv = dot(V, V);
    double pp = dot(P, P);
    double dd = dot(D, D);

    double dv = dot(D, V);
    double dp = dot(D, P);
    double pv = dot(P, V);

    double a = dd * vv -  dv * dv;
    double b = 2.0 * (dd * pv - dp * dv);
    double c = dd * pp - dp * dp - r*r * dd;

    double d = b*b - 4*a*c;
    if (d >= 0) {
        d = sqrt(d);
        double t1 = (-b-d)/(2.0*a);
        double t2 = (-b+d)/(2.0*a);
        double t1p = dot(P + V*t1, D)/dd;
        double t2p = dot(P + V*t2, D)/dd;
        if (t1 > tMin && t1 < tMax && t1p >= 0 && t1p <= 1)  {
            inter.t = t1;
            inter.pos = ray.pos + ray.dir * t1;
            inter.normal = inter.pos - (A + D * t1p);
            inter.normal.normalize();
            if (dot(ray.dir, inter.normal) >= 0)
                inter.normal = -inter.normal;
```

```
            return true;
        } else
        if (t2 > tMin && t2 < tMax && t2p >= 0 && t2p <= 1)  {
            inter.t = t2;
            inter.pos = ray.pos + ray.dir * t2;
            inter.normal = inter.pos - (A + D * t2p);
            inter.normal.normalize();
            if (dot(ray.dir, inter.normal) >= 0)
                inter.normal = -inter.normal;
            return true;
        } else
            return false;
    } else
        return false;
}

};
```

FILE **image.cpp**

```
#include <fstream>
#include <algorithm>
#include <pt/image.h>

namespace pt
{

//---------------------------------------------------------------------------
Image::Image(int width, int height) {
    m_pix = new Color[width * height];
    m_width = width;
    m_height = height;
}

//---------------------------------------------------------------------------
Image::Image(const Image& img) {
    m_pix = new Color[img.m_width * img.m_height];
    m_width = img.m_width;
    m_height = img.m_height;
    for (int i = 0; i < m_width * m_height; ++i) {
        m_pix[i] = img.m_pix[i];
    }
}

//---------------------------------------------------------------------------
Image::Image(const Image * img) {
    try {
        m_pix = new Color[img->m_width * img->m_height];
        m_width = img->m_width;
        m_height = img->m_height;
        for (int i = 0; i < m_width * m_height; ++i) {
            m_pix[i] = img->m_pix[i];
        }
    } catch(...) {
        m_pix = new Color[1000 * 1000];
        m_width = 1000;
        m_height = 1000;
        for (int i = 0; i < m_width * m_height; ++i) {
            m_pix[i] = Color(0, 0, 0);
        }
    }
}

//---------------------------------------------------------------------------
Image::~Image() {
    delete[] m_pix;
}

//---------------------------------------------------------------------------
void Image::resize(int width, int height) {
    delete[] m_pix;
    m_pix = new Color[width * height];
    m_width = width;
    m_height = height;
}
```

```cpp

//------------------------------------------------------------------------------
void Image::clear(void) {
    for (int i = 0; i < m_width * m_height; ++i) {
        m_pix[i] = Color(0, 0, 0, 0);
    }
}

//------------------------------------------------------------------------------
void Image::colorCorrection(void) {
    for (int i = 0; i < m_width * m_height; ++i) {
        m_pix[i] = m_pix[i].sqrt();
        if (m_pix[i].a > 1) m_pix[i].a = 1;
        if (m_pix[i].r > 1) m_pix[i].r = 1;
        if (m_pix[i].g > 1) m_pix[i].g = 1;
        if (m_pix[i].b > 1) m_pix[i].b = 1;
    }
}

//------------------------------------------------------------------------------
int Image::getWidth() const {
    return m_width;
}

//------------------------------------------------------------------------------
int Image::getHeight() const {
    return m_height;
}

//------------------------------------------------------------------------------
Color& Image::operator()(int x, int y) {
    return m_pix[x + y * m_width];
}

//------------------------------------------------------------------------------
const Color& Image::operator()(int x, int y) const {
    return m_pix[x + y * m_width];
}

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
void saveAsDoubleImg(const Image& img, const std::string& name) {
    std::ofstream fout(name, std::ios::binary);
    double value = 0;
    value = img.getWidth(); fout.write(reinterpret_cast<const char*>(&value), sizeof(double));
    value = img.getHeight(); fout.write(reinterpret_cast<const char*>(&value), sizeof(double));
    for (int i = 0; i < img.getWidth(); i++) {
        for (int j = 0; j < img.getHeight(); j++) {
            value = img(i, j).r; fout.write(reinterpret_cast<const char*>(&value), sizeof(double));
        }
    }
    fout.close();
}

//------------------------------------------------------------------------------
void loadAsDoubleImg(Image& img, const std::string& name) {
    std::ifstream fin(name, std::ios::binary);
    double value = 0, width, height;
    fin.read(reinterpret_cast<char*>(&value), sizeof(double)); width = value;
    fin.read(reinterpret_cast<char*>(&value), sizeof(double)); height = value;
    img.resize(width, height);
    for (int i = 0; i < img.getWidth(); i++) {
        for (int j = 0; j < img.getHeight(); j++) {
            fin.read(reinterpret_cast<char*>(&value), sizeof(double)); img(i, j).r = value;
            img(i, j).g = 0;
            img(i, j).b = 0;
            img(i, j).a = 1;
        }
    }
    fin.close();
}

```

```
//-----------------------------------------------------------------------------
void toGrayScaleDoubleImg(Image& img, double ignoreMax) {
    double mymin = img(0, 0).r, mymax = 0;

    for (int i = 0; i < img.getWidth(); i++) {
        for (int j = 0; j < img.getHeight(); j++) {
            mymin = std::min(mymin, img(i, j).r);
            if (ignoreMax < 0 || img(i, j).r < ignoreMax)
                mymax = std::max(mymax, img(i, j).r);
        }
    }

    for (int i = 0; i < img.getWidth(); i++) {
        for (int j = 0; j < img.getHeight(); j++) {
            double value = (img(i, j).r - mymin)/(mymax-mymin);
            if (value > 1) value = 1;
            value = 1 - value;
            img(i, j).r = value;
            img(i, j).g = value;
            img(i, j).b = value;
            img(i, j).a = 1;
        }
    }
}

};
```

FILE **light.cpp**

```
#include <pt/material/light.h>

namespace pt
{

//-----------------------------------------------------------------------------
Light::Light(Color clr) : clr(clr) {

}

//-----------------------------------------------------------------------------
ScatterType Light::scatter(const Ray& ray,
                           const Intersection& inter,
                           Color& clrAbsorbtion,
                           Ray& scattered,
                           double& diffusion) const {
    clrAbsorbtion = clr;
    diffusion = 0;
    return SCATTER_END;
}

};
```

FILE **orthogonal.cpp**

```
#include <pt/camera/orthogonal.h>

namespace pt
{

//-----------------------------------------------------------------------------
Orthogonal::Orthogonal(vec3 pos, double scale, double width, double height) : Camera(pos),
↪  scale(scale), width(width), height(height) {
}

//-----------------------------------------------------------------------------
Ray Orthogonal::getRay(double x, double y, bool isDiffuse) const {
    x -= width/2.0;
    y -= height/2.0;
    x *= scale;
    y *= scale;
    Ray ray;
    ray.pos = -i*x - j*y + pos;
    ray.dir = k;
    return ray;
}
```

```

//------------------------------------------------------------------------
void Orthogonal::lookTowards(vec3 toward) {
    k = toward - pos;
    k.normalize();
    if (!(k.x == 0 && k.y == 0)) {
        i.x = -k.y;
        i.y = k.x;
        i.normalize();
    }
    j = cross(k, i);
    i = -i;
    j.normalize();
}

};
```

FILE **poly.cpp**

```
#include <pt/poly.h>

using namespace spob;

//------------------------------------------------------------------------
/** Big thanks to M. Galetzka and P. Glauner. This code was copied from
↪  https://github.com/pglauner/point_in_polygon . */

//------------------------------------------------------------------------
struct Line {
    vec2 p1;
    vec2 p2;
};

//------------------------------------------------------------------------
inline int ccw(const vec2& p0, const vec2& p1, const vec2& p2) {
    double dx1 = p1.x - p0.x;
    double dy1 = p1.y - p0.y;
    double dx2 = p2.x - p0.x;
    double dy2 = p2.y - p0.y;
    double dx1dy2 = dx1 * dy2;
    double dx2dy1 = dx2 * dy1;

    if (dx1dy2 > dx2dy1) {
        return 1;
    } else if (dx1dy2 < dx2dy1) {
        return -1;
    }
    else {
        if (dx1 * dx2 < 0 || dy1 * dy2 < 0) {
            return -1;
        } else if (dx1 * dx1 + dy1 * dy1 >= dx2 * dx2 + dy2 * dy2) {
            return 0;
        } else {
            return 1;
        }
    }
}

//------------------------------------------------------------------------
inline int intersect(const Line& line1, const Line& line2) {
    int ccw11 = ccw(line1.p1, line1.p2, line2.p1); if (ccw11 == 0) return 1;
    int ccw12 = ccw(line1.p1, line1.p2, line2.p2); if (ccw12 == 0) return 1;
    if (!(ccw11 * ccw12 < 0))
        return 0;
    int ccw21 = ccw(line2.p1, line2.p2, line1.p1); if (ccw21 == 0) return 1;
    int ccw22 = ccw(line2.p1, line2.p2, line1.p2); if (ccw22 == 0) return 1;
    return (ccw21 * ccw22 < 0) ? 1 : 0;
}

//------------------------------------------------------------------------
inline int getNextIndex(int n, int current) {
    return current == n - 1 ? 0 : current + 1;
}

//------------------------------------------------------------------------
```

```
bool pointInPolygon(const std::vector<vec2>& polygon1, vec2 testPoint) {
    auto polygon = polygon1;
    int n = polygon.size();
    Line xAxis;
    Line xAxisPositive;

    vec2 startPoint;
    vec2 endPoint;
    Line edge;
    Line testPointLine;

    int i;
    double startNodePosition;
    int count;
    int seenPoints;

    // Initial start point
    startPoint.x = 0;
    startPoint.y = 0;

    // Create axes
    xAxis.p1.x = 0;
    xAxis.p1.y = 0;
    xAxis.p2.x = 0;
    xAxis.p2.y = 0;
    xAxisPositive.p1.x = 0;
    xAxisPositive.p1.y = 0;
    xAxisPositive.p2.x = 0;
    xAxisPositive.p2.y = 0;

    startNodePosition = -1;

    // Is testPoint on a node?
    // Move polygon to 0|0
    // Enlarge axes
    for (i = 0; i < n; i++) {
        if (testPoint.x == polygon[i].x && testPoint.y == polygon[i].y) {
            return 1;
        }

        // Move polygon to 0|0
        polygon[i].x -= testPoint.x;
        polygon[i].y -= testPoint.y;

        // Find start point which is not on the x axis
        if (polygon[i].y != 0) {
            startPoint.x = polygon[i].x;
            startPoint.y = polygon[i].y;
            startNodePosition = i;
        }

        // Enlarge axes
        if (polygon[i].x > xAxis.p2.x) {
            xAxis.p2.x = polygon[i].x;
            xAxisPositive.p2.x = polygon[i].x;
        }
        if (polygon[i].x < xAxis.p1.x) {
            xAxis.p1.x = polygon[i].x;
        }
    }

    // Move testPoint to 0|0
    testPoint.x = 0;
    testPoint.y = 0;
    testPointLine.p1 = testPoint;
    testPointLine.p2 = testPoint;

    // Is testPoint on an edge?
    for (i = 0; i < polygon.size(); i++) {
        edge.p1 = polygon[i];
        // Get correct index of successor edge
        edge.p2 = polygon[getNextIndex(polygon.size(), i)];
        if (intersect(testPointLine, edge) == 1) {
            return 1;
        }
```

```
    }

    // No start point found and point is not on an edge or node
    // --> point is outside
    if (startNodePosition == -1) {
        return 0;
    }

    count = 0;
    seenPoints = 0;
    i = startNodePosition;

    // Consider all edges
    while (seenPoints < n) {

        double savedX = polygon[getNextIndex(n, i)].x;
        int savedIndex = getNextIndex(n, i);

        // Move to next point which is not on the x-axis
        do {
            i = getNextIndex(n, i);
            seenPoints++;
        } while (polygon[i].y == 0);
        // Found end point
        endPoint.x = polygon[i].x;
        endPoint.y = polygon[i].y;

        // Only intersect lines that cross the x-axis
        if (startPoint.y * endPoint.y < 0) {
            edge.p1 = startPoint;
            edge.p2 = endPoint;

            // No nodes have been skipped and the successor node
            // has been chosen as the end point
            if (savedIndex == i) {
                count += intersect(edge, xAxisPositive);
            }
            // If at least one node on the right side has been skipped,
            // the original edge would have been intersected
            // --> intersect with full x-axis
            else if (savedX > 0) {
                count += intersect(edge, xAxis);
            }
        }
        // End point is the next start point
        startPoint = endPoint;
    }

    // Odd count --> in the polygon (1)
    // Even count --> outside (0)
    return count % 2;
}
```

FILE **polygon.cpp**

```
#include <pt/shape/polygon.h>

namespace pt
{

//-----------------------------------------------------------------------------
Polygon::Polygon(const std::vector<vec2>& polygon, crd3 coords1, Material_ptr material) :
↪   Shape(material) {
    assign(polygon, coords1, material);
}

//-----------------------------------------------------------------------------
void Polygon::assign(const std::vector<vec2>& polygon, crd3 coords1, Material_ptr material1) {
    array = polygon;
    coords = coords1;
    material = material1;

    d = -dot(coords.k, coords.pos);
    normal = coords.k;
    normal = normal.normalize();
```

```

    min.x = +std::numeric_limits<double>::infinity();
    min.y = +std::numeric_limits<double>::infinity();
    max.x = -std::numeric_limits<double>::infinity();
    max.y = -std::numeric_limits<double>::infinity();

    for (size_t i = 0; i < polygon.size(); i++) {
        if (polygon[i].x < min.x) min.x = polygon[i].x;
        if (polygon[i].y < min.y) min.y = polygon[i].y;
        if (polygon[i].x > max.x) max.x = polygon[i].x;
        if (polygon[i].y > max.y) max.y = polygon[i].y;
    }
}

//-----------------------------------------------------------------------------
bool Polygon::intersect(const Ray& ray,
                        Intersection& inter,
                        double tMin,
                        double tMax) const {
    if (dot(ray.dir, normal) != 0) {
        double t = (-d - dot(ray.pos, normal))/dot(ray.dir, normal);

        if (t > tMin && t < tMax) {
            // Position when ray intersect plane
            vec3 x = ray.pos + ray.dir * t;

            vec2 r;
            r.x = (dot(x, coords.i) - dot(coords.pos, coords.i))/dot(coords.i, coords.i);
            r.y = (dot(x, coords.j) - dot(coords.pos, coords.j))/dot(coords.j, coords.j);

            bool inRect = r.x >= min.x && r.y >= min.y && r.x <= max.x && r.y <= max.y;
            if (!inRect)
                return false;

            if (pointInPolygon(array, r)) {
                inter.t = t;
                if (dot(ray.dir, normal) < 0)
                    inter.normal = normal;
                else
                    inter.normal = -normal;
                inter.pos = x;
                return true;
            } else
                return false;
        } else
            // Ray is not in ranges
            return false;
    } else {
        // Ray is parallel plane
        return false;
    }
}

};
```

FILE **portals.cpp**

```
#include <pt/shape/portals.h>

namespace pt
{

//-----------------------------------------------------------------------------
Portals::Portals(crd3 c1, crd3 c2, std::vector<vec2> poly, Material_ptr first, Material_ptr second) :
↪  p1(c1), p2(c2), poly(poly), first(first), second(second), pg1(poly, c1, nullptr), pg2(poly, c2,
↪  nullptr) {
}

//-----------------------------------------------------------------------------
Portals::~Portals() {
}

//-----------------------------------------------------------------------------
void Portals::assign(crd3 c1, crd3 c2, std::vector<vec2> poly1, Material_ptr first1, Material_ptr
↪  second1) {
```

```
    pg1.assign(poly1, c1, nullptr);
    pg2.assign(poly1, c2, nullptr);
    p1 = c1;
    p2 = c2;
    poly = poly1;
    first = first1;
    second = second1;
}

//-----------------------------------------------------------------------------
bool Portals::intersect(const Ray& ray,
                        Intersection& inter,
                        double tMin,
                        double tMax) const {
    Intersection inter1, inter2;
    bool isFirst = pg1.intersect(ray, inter1, tMin, tMax);
    bool isSecond = pg2.intersect(ray, inter2, tMin, tMax);

    isFirst &= inter1.t >= tMin && inter1.t <= tMax;
    isSecond &= inter2.t >= tMin && inter2.t <= tMax;
    bool firstLessSecond = inter1.t < inter2.t;

    if (isFirst && isSecond && firstLessSecond) {
        inter = inter1;
        inter.data.integer = 1;
        return true;
    }

    if (isFirst && isSecond && !firstLessSecond) {
        inter = inter2;
        inter.data.integer = 2;
        return true;
    }

    if (isFirst) {
        inter = inter1;
        inter.data.integer = 1;
        return true;
    }

    if (isSecond) {
        inter = inter2;
        inter.data.integer = 2;
        return true;
    }

    return false;
}

//-----------------------------------------------------------------------------
ScatterType Portals::scatter(const Ray& ray,
                             const Intersection& inter,
                             Color& clrAbsorbtion,
                             Ray& scattered,
                             double& diffusion) const {
    if (inter.data.integer == 1)
        if (dot(inter.normal, p1.k) > 0)
            return first->scatter(ray, inter, clrAbsorbtion, scattered, diffusion);
    if (inter.data.integer == 2)
        if (dot(inter.normal, p2.k) < 0)
            return second->scatter(ray, inter, clrAbsorbtion, scattered, diffusion);

    scattered.pos = inter.pos;
    scattered.dir = ray.dir;
    if (inter.data.integer == 1) {
        scattered.pos = p2.from(p1.to(scattered.pos));
        scattered.dir = p2.fromDir(p1.toDir(scattered.dir));
    } else {
        scattered.pos = p1.from(p2.to(scattered.pos));
        scattered.dir = p1.fromDir(p2.toDir(scattered.dir));
    }
    clrAbsorbtion = Color(1, 1, 1, 1);
    diffusion = 0;
    return SCATTER_NEXT;
}
```

```
91
92 //-----------------------------------------------------------------------------
93 Portals invert(Portals a) {
94     crd3 coords1 = a.p2;
95     coords1.k = -coords1.k;
96     crd3 coords2 = a.p1;
97     coords2.k = -coords2.k;
98     return Portals(coords1, coords2, a.poly, a.second, a.first);
99 }
100
101 }
```

FILE **pt2easybmp.cpp**

```
1  #define STB_IMAGE_IMPLEMENTATION
2  #define STB_IMAGE_WRITE_IMPLEMENTATION
3  #define STBI_MSC_SECURE_CRT
4  #include <stb_image.h>
5  #include <stb_image_write.h>
6
7  #include <pt/pt2easybmp.h>
8
9  namespace pt
10 {
11
12 //-----------------------------------------------------------------------------
13 void saveAsPng(Image& img, std::string name) {
14     unsigned char *data = new unsigned char[img.getWidth() * img.getHeight() * 4];
15     for (int i = 0; i < img.getHeight(); i++) {
16         for (int j = 0; j < img.getWidth(); j++) {
17             int offset = 4*(i * img.getWidth() + j);
18             auto clr = img(j, i);
19             data[offset + 0] = clr.r * 255;
20             data[offset + 1] = clr.g * 255;
21             data[offset + 2] = clr.b * 255;
22             data[offset + 3] = clr.a * 255;
23         }
24     }
25     stbi_write_png_compression_level = 64;
26     stbi_write_png(name.c_str(), img.getWidth(), img.getHeight(), 4, data, img.getWidth() * 4);
27     delete[] data;
28 }
29
30 //-----------------------------------------------------------------------------
31 void loadAsPng(Image& img, std::string name) {
32     unsigned char *data;
33     int width, height, n;
34     data = stbi_load(name.c_str(), &width, &height, &n, 4);
35     img.resize(width, height);
36     for (int i = 0; i < height; ++i) {
37         for (int j = 0; j < width; ++j) {
38             img(j, i) = Color(
39                 data[4*(i * width + j)+0]/255.0,
40                 data[4*(i * width + j)+1]/255.0,
41                 data[4*(i * width + j)+2]/255.0,
42                 data[4*(i * width + j)+3]/255.0
43             );
44         }
45     }
46     free(data);
47 }
48
49
50 };
```

FILE **reflect.cpp**

```
1 #include <pt/material/reflect.h>
2
3 namespace pt
4 {
5
6 //-----------------------------------------------------------------------------
7 Reflect::Reflect(Color clr, double diffusion) : clr(clr), diffuse(diffusion) {
8 }
```

```
9
10 //-----------------------------------------------------------------------------
11 ScatterType Reflect::scatter(const Ray& ray,
12                              const Intersection& inter,
13                              Color& clrAbsorbtion,
14                              Ray& scattered,
15                              double& diffusion) const {
16     clrAbsorbtion = clr;
17     scattered.pos = inter.pos;// + inter.normal * 0.001;
18     scattered.dir = ray.dir;
19     reflect(scattered.dir, inter.normal);
20     diffusion = diffuse;
21     return SCATTER_NEXT;
22 }
23
24 };
```

FILE **refract.cpp**

```
1  #include <pt/material/refract.h>
2
3  namespace pt
4  {
5
6  //-----------------------------------------------------------------------------
7  Refract::Refract(double refractiveIndex, double diffusion) : refractiveIndex(refractiveIndex),
   ↪  diffuse(diffusion) {
8
9  }
10
11 //-----------------------------------------------------------------------------
12 ScatterType Refract::scatter(const Ray& ray,
13                              const Intersection& inter,
14                              Color& clrAbsorbtion,
15                              Ray& scattered,
16                              double& diffusion) const {
17     double ri;
18     vec3 normal;
19
20     if (dot(ray.dir, inter.normal) > 0) {
21         // Out of object
22         ri = refractiveIndex;
23         normal = -inter.normal;
24     } else {
25         // In object
26         ri = 1 / refractiveIndex;
27         normal = inter.normal;
28     }
29
30     scattered.dir = ray.dir;
31     if (!refract(scattered.dir, normal, ri)) {
32         scattered.dir = ray.dir;
33         reflect(scattered.dir, normal);
34     }
35
36     scattered.pos = inter.pos;// + scattered.dir * 0.001;
37     diffusion = diffuse;
38     clrAbsorbtion = Color(1, 1, 1, 1);
39
40     return SCATTER_NEXT;
41 }
42
43 };
```

FILE **renderer.cpp**

```
1 #include <sstream>
2 #include <algorithm>
3 #include <random>
4 #include <iostream>
5 #include <stack>
6 #include <chrono>
7 #include <string>
8 #include <iomanip>
9 #include <mutex>
```

```
10
11 #include <pt/renderer.h>
12
13 namespace pt
14 {
15
16 namespace {
17     typedef std::chrono::high_resolution_clock hrc;
18
19     //-------------------------------------------------------------------------
20     float getCurrentTime(void) {
21         static hrc::time_point t = hrc::now();
22         return std::chrono::duration<double>(hrc::now() - t).count();
23     }
24
25     //-------------------------------------------------------------------------
26     float getTimePassed(float pastTime) {
27         return getCurrentTime() - pastTime;
28     }
29
30     //-------------------------------------------------------------------------
31     float getApproxTime(float pastTime, float percent) {
32         if (percent == 0)
33             return 0;
34         else
35             return getTimePassed(pastTime)/percent;
36     }
37
38     //-------------------------------------------------------------------------
39     float getLeftTime(float pastTime, float percent) {
40         if (percent == 0)
41             return 0;
42         else
43             return getApproxTime(pastTime, percent) - getTimePassed(pastTime);
44     }
45
46     //-------------------------------------------------------------------------
47     std::string getTimeString(float time) {
48         char s[25] = {};
49         if (true) {
50             if (time > 86400)
51                 sprintf(s, "%2dd %2dh %2dm %2ds", int(time/86400), int(time/3600)%24,
                    ↪  int(time/60)%60, int(time)%60);
52             else
53             if (time > 3600)
54                 sprintf(s, "    %2dh %2dm %2ds", int(time/3600)%24, int(time/60)%60, int(time)%60);
55             else
56             if (time > 60)
57                 sprintf(s, "        %2dm %2ds", int(time/60)%60, int(time)%60);
58             else
59                 sprintf(s, "            %2ds", int(time)%60);
60         } else {
61             if (time > 86400)
62                 sprintf(s, "%2dd %2dh %2dm %2ds", int(time/86400), int(time/3600)%24,
                    ↪  int(time/60)%60, int(time)%60);
63             else
64             if (time > 3600)
65                 sprintf(s, "%2dh %2dm %2ds", int(time/3600)%24, int(time/60)%60, int(time)%60);
66             else
67             if (time > 60)
68                 sprintf(s, "%2dm %2ds", int(time/60)%60, int(time)%60);
69             else
70                 sprintf(s, "%2ds", int(time)%60);
71         }
72         return std::string(s);
73     }
74 }
75
76 //-------------------------------------------------------------------------
77 //-------------------------------------------------------------------------
78 //-------------------------------------------------------------------------
79
80 //-------------------------------------------------------------------------
81 StandardRenderer::StandardRenderer(
82     int threads,
```

```
    int maxDepth,
    double tMax,
    bool isDiffuse,
    bool isBreakOnMaterial,
    bool isWriteText) :
    threads(threads),
    maxDepth(maxDepth),
    tMax(tMax),
    isDiffuse(isDiffuse),
    isBreakOnMaterial(isBreakOnMaterial),
    isWriteText(isWriteText),
    ambient(0, 0, 0, 1){
}

//-----------------------------------------------------------------------------
StandardRenderer::~StandardRenderer() {
}

//-----------------------------------------------------------------------------
void StandardRenderer::addPortal(Portals_ptr portal) {
    portals.push_back(portal);
    Portals_ptr inv = std::make_shared<Portals>(invert(*portal));
    invertedPortals.push_back(inv);
}

//-----------------------------------------------------------------------------
void StandardRenderer::clearPortals(void) {
    portals.clear();
    invertedPortals.clear();
}

//-----------------------------------------------------------------------------
void StandardRenderer::render(void) {
    std::vector<int> pixels(img->getWidth() * img->getHeight(), 0);
    for (int i = 0; i < pixels.size(); ++i)
        pixels[i] = i;

    static std::random_device rd;
    static std::mt19937 g(rd());

    std::shuffle(pixels.begin(), pixels.end(), g);

    std::mutex write_mutex;
    int renderedPixelsCount = 0;

    onStartRender();
    std::vector<std::thread> threads_mas;
    for (int i = 0; i < threads; ++i) {
        threads_mas.push_back(std::thread([&] (int i) {
            for (int j = i; j < pixels.size(); j += threads) {
                write_mutex.lock();
                renderedPixelsCount++;
                if (renderedPixelsCount % img->getHeight() == 0)
                    onEveryLine(double(renderedPixelsCount)/pixels.size());
                write_mutex.unlock();

                int x = pixels[j] % img->getWidth();
                int y = pixels[j] / img->getWidth();
                auto pix = computePixel(x, y);
                (*img)(x, y) = pix.color;
                if (isDrawDepth)
                    (*dImg)(x, y) = Color(pix.depth, 0, 0);
            }
        }, i));
    }
    for (auto& i : threads_mas)
        i.join();
    onEndRendering();
}

//-----------------------------------------------------------------------------
void StandardRenderer::onStartRender(void) {
    using namespace std;
    if (isWriteText) {
        time = getCurrentTime();
```

```
158         cout << setfill(' ') << setiosflags(ios_base::right);
159         cout << "Percent |" << setw(23) << "Time passed |" << setw(23) << "Approximate time |" <<
            ↪  setw(24) << "Time Left |" << endl;
160         cout << setfill('-') << setw(79) << '|' << endl;
161         cout << setfill(' ');
162     }
163 }
164
165 //--------------------------------------------------------------------------
166 void StandardRenderer::onEveryLine(double percent) const {
167     using namespace std;
168     if (isWriteText) {
169         stringstream sout;
170
171         cout << '\r';
172
173         sout.str(std::string());
174         sout << setprecision(2) << percent * 100 << "% |";
175         cout << setw(9) << sout.str();
176
177         sout.str(std::string());
178         sout << getTimeString(getTimePassed(time)) << " |";
179         cout << setw(23) << sout.str();
180
181         sout.str(std::string());
182         sout << getTimeString(getApproxTime(time, percent)) << " |";
183         cout << setw(23) << sout.str();
184
185         sout.str(std::string());
186         sout << getTimeString(getLeftTime(time, percent)) << " |";
187         cout << setw(24) << sout.str();
188     }
189 }
190
191 //--------------------------------------------------------------------------
192 void StandardRenderer::onEndRendering(void) const {
193     using namespace std;
194     if (isWriteText) {
195         cout << '\r' << setw(9) << "100% |";
196
197         stringstream sout;
198         sout.clear();
199         sout << getTimeString(getTimePassed(time)) << " |";
200         cout << setw(23) << sout.str();
201
202         cout << setw(23) << "0s |" << setw(24) << "0s |" << endl;
203         cout << endl;
204     }
205 }
206
207 //--------------------------------------------------------------------------
208 Renderer1::Frag StandardRenderer::computePixel(int x, int y) const {
209     return computeColor(camera->getRay(x, y, isDiffuse));
210 }
211
212 //--------------------------------------------------------------------------
213 Renderer1::Frag StandardRenderer::computeColor(Ray ray) const {
214     Intersection inter;
215     Color clrAbsorbtion;
216     std::stack<Color> colorStack;
217     std::stack<Color> pointLightColorStack;
218     std::stack<bool> pointLightColorStackBool;
219     Ray scattered;
220     double diffusion;
221     ScatterType returned;
222
223     double depth = tMax;
224     bool isDepthInitialized = false;
225
226     for (int i = 0; i < maxDepth; ++i) {
227         if (scene->intersect(ray, inter, 0, tMax)) {
228             if (!isDepthInitialized) {
229                 depth = inter.t;
230                 isDepthInitialized = true;
231             }
```

```
232
233             returned = scene->scatter(ray, inter, clrAbsorbtion, scattered, diffusion);
234
235             // Запоминаем прозрачность текущего цвета
236             double opaque = clrAbsorbtion.a;
237             clrAbsorbtion.a = 1;
238
239             // Изменить направление в соответствии с рассеиванием
240             if (isDiffuse)
241                 scattered.dir += randomSphere() * diffusion;
242             scattered.dir.normalize();
243
244             // Сместить положение луча в некотором направлении
245             scattered.pos += scattered.dir * 0.00001;
246
247             // Считаем цвет освещения, но его надо считать только когда у нас обычный материал
248             if (returned == SCATTER_RAYTRACING_END && luminaries.size() != 0) {
249                 Color lightColor = Color(0, 0, 0, 1);
250                 std::vector<std::pair<Portals_ptr, vec3> > teleportation;
251                 lightColor += computeLight(scattered.pos, inter.normal, teleportation, 3);
252                 pointLightColorStack.push(lightColor * clrAbsorbtion);
253                 pointLightColorStackBool.push(true);
254             } else {
255                 pointLightColorStack.push(Color(0, 0, 0, 0));
256                 pointLightColorStackBool.push(false);
257             }
258
259             // Посчитать результирующй цвет после данного отражения
260             colorStack.push(clrAbsorbtion);
261
262             // Если полигон полупрозрачный, то его цвет будет комбинацией двух лучей, сложенных с
                ↪  учетом прозрачности
263             if (opaque != 1.0) {
264                 // Отложим поддержку прозрачности до лучших времен
265                 /*// Изменяем и сохраняем глубину рендерера, чтобы он не зациклился
266                 int temp = maxDepth;
267                 maxDepth -= i + 1;
268
269                 // Получаем цвет луча, который пойдет сквозь материал
270                 Color throughColor;
271                 if (returned != SCATTER_END) {
272                     Ray through;
273                     through.pos = scattered.pos;
274                     through.dir = ray.dir;
275                     through.pos += through.dir * 0.00001;
276                     throughColor = computeColor(through);
277                 }
278
279                 // Получаем цвет луча, который пошел бы обычным путем
280                 Color nextColor;
281                 if (!(returned == SCATTER_RAYTRACING_END && isBreakOnMaterial))
282                     nextColor = computeColor(scattered);
283
284                 // Совмещаем два цвета с учетом прозрачности
285                 resultColor = resultColor * clrAbsorbtion;
286                 resultColor.a = opaque;
287                 resultColor = overlay(resultColor, throughColor);
288
289                 maxDepth = temp;
290                 break;*/
291             }
292
293             ray = scattered;
294             if (returned == SCATTER_END || (returned == SCATTER_RAYTRACING_END && isBreakOnMaterial))
295                 break;
296         } else {
297             if (i == 0)
298                 return Frag(Color(0, 0, 0, 0), depth);
299             else
300                 break;
301         }
302     }
303
304     Color resultColor(1, 1, 1, 1);
305     if (returned == SCATTER_END) {
```

```
306         resultColor = colorStack.top();
307 
308         colorStack.pop();
309         pointLightColorStack.pop();
310         pointLightColorStackBool.pop();
311     } else {
312         resultColor = ambient;
313     }
314     while (!colorStack.empty()) {
315         if (pointLightColorStackBool.top()) {
316             resultColor = colorStack.top() * resultColor;
317             resultColor += pointLightColorStack.top();
318             resultColor /= 2.0;
319         } else {
320             resultColor = colorStack.top() * resultColor;
321         }
322 
323         colorStack.pop();
324         pointLightColorStack.pop();
325         pointLightColorStackBool.pop();
326     }
327 
328     return Frag(resultColor, depth);
329 }
330 
331 //------------------------------------------------------------------------------
332 Color StandardRenderer::computeLight(
333     vec3 pos, vec3 normal,
334     std::vector<std::pair<Portals_ptr, vec3> >& teleportation,
335     int depth) const {
336     // В этой функции предполагается, что все источники света должны быть телепортированы через
      ↪  порталы, указанные в teleportation, и для всех них как раз проверяется, чтобы через все эти
      ↪  порталы свет мог попасть к текущему месту, которое проверяется на освещенность
337 
338     // Тут тоже отложим поддержку прозрачности до лучших времен
339     Color result(0, 0, 0, 0);
340     Intersection inter;
341     double t;
342     double cosine;
343     bool isIntersect;
344     Ray ray;
345 
346     for (auto i : luminaries) {
347         // Эта переменная отвечает за то, чтобы луч света проходил через все текущие порталы, если
          ↪  она на миг становится false, то перебор данного источника света сразу прекращается
348         bool isPass = true;
349         for (int j = teleportation.size() - 1; j >= 0; --j) {
350             // Получаем луч, который идет от текущего телепортированного положения до текущего
              ↪  источника света
351             vec3& pos = teleportation[j].second;
352             Portals& portal = *teleportation[j].first;
353             ray = {pos, i.pos - pos};
354             ray.dir.normalize();
355 
356             // Проверяем, чтобы этот луч входил в портал
357             isPass &= dot(ray.dir, portal.p2.k) > 0;
358             if (!isPass) goto next;
359             isPass &= portal.pg2.intersect(ray, inter, 0, tMax);
360             if (!isPass) goto next;
361 
362             // Проверяем, чтобы на пути от портала до текущего положения источника света не было
              ↪  препятствий
363             ray.pos += ray.dir * (inter.t + 0.00001);
364             t = distance(i.pos, ray.pos);
365             isIntersect = scene->intersect(ray, inter, 0, t + 1);
366             isPass &= !isIntersect || (isIntersect && inter.t > t);
367             if (!isPass) goto next;
368 
369             // Сдвигаем источник света по лучу ближе к текущему месту, на освещенность данного
              ↪  конкретного места не повлияет, а после сдвига телепортируем, чтобы рассчитывать это
              ↪  для других порталов
370             i.pos -= ray.dir * (t + 0.00003);
371             i.pos = portal.p1.from(portal.p2.to(i.pos));
372         }
373 
```

```
        // Получаем луч от текущего абсолютного места до текущего источника света
        ray = {pos, i.pos - pos};
        ray.dir.normalize();
        normal.normalize();
        ray.pos += ray.dir * 0.00001;

        // Проверяем, чтобы на пути не было препятствий
        t = distance(i.pos, pos);
        isIntersect = scene->intersect(ray, inter, 0, t + 1);
        isPass &= !isIntersect || (isIntersect && inter.t > t);
        if (!isPass) goto next;

        // Если всех этих препятствий нет, то прибавляем освещение от этого источника света
        cosine = dot(ray.dir, normal);
        result += i.clr * cosine;

        next:;
    }

    // Далее, если позволяет глубина, перебираем все порталы дальше
    if (depth > 0) {
        for (int i = 0; i < portals.size(); ++i) {
            auto recursion = [&] (Portals_ptr portal) {
                vec3 newPos;
                if (teleportation.size() != 0)
                    newPos = portal->p2.from(portal->p1.to(teleportation.back().second));
                else
                    newPos = portal->p2.from(portal->p1.to(pos));
                teleportation.push_back({portal, newPos});
                result += computeLight(pos, normal, teleportation, depth-1);
                teleportation.pop_back();
            };

            // Перебираем прямой и обратный порядок следования порталов
            recursion(portals[i]);
            recursion(invertedPortals[i]);
        }
    }
    return result;
}

//------------------------------------------------------------------------------
void StandardRenderer::setAmbientLight(Color clr) {
    ambient = clr;
}

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
RayTracing::RayTracing(int aliasing,
                       int threads,
                       bool isWriteText,
                       int maxDepth,
                       double tMax) : StandardRenderer(threads, maxDepth, tMax, false, true,
                       ↪  isWriteText), antialiasing(aliasing) {
}

//------------------------------------------------------------------------------
Renderer1::Frag RayTracing::computePixel(int x, int y) const {
    Color clr(0, 0, 0, 0);
    double depth = tMax;
    for (int ki = 0; ki < antialiasing; ++ki) {
        for (int kj = 0; kj < antialiasing; ++kj) {
            double x1 = x + double(ki)/antialiasing;
            double y1 = y + double(kj)/antialiasing;
            Ray ray = camera->getRay(x1, y1, isDiffuse);
            auto pix = computeColor(ray);
            clr += pix.color;
            depth = std::min(depth, pix.depth);
        }
    }
    clr /= antialiasing * antialiasing;
    return Frag(clr, depth);
```

```
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
PathTracing::PathTracing(int samples,
                         int threads,
                         bool isWriteText,
                         int maxDepth,
                         double tMax) : StandardRenderer(threads, maxDepth, tMax, true, false,
                         ↪  isWriteText), samples(samples) {
}

//-----------------------------------------------------------------------------
Renderer1::Frag PathTracing::computePixel(int x, int y) const {
    // Использована квазислучайная последовательность https://habr.com/ru/post/440892/
    const double g = 1.32471795724474602596090885447809;
    const double ax = std::fmod(1.0/g, 1.0);
    const double ay = std::fmod(1.0/g/g, 1.0);
    const double seed = 0.5;

    Color clr(0, 0, 0, 0);
    double depth = tMax;
    for (int i = 0; i < samples; ++i) {
        double x1 = x + std::fmod(seed + ax*i, 1.0);
        double y1 = y + std::fmod(seed + ay*i, 1.0);
        Ray ray = camera->getRay(x1, y1, isDiffuse);
        auto pix = computeColor(ray);
        clr += pix.color;
        depth = std::min(depth, pix.depth);
    }
    clr /= samples;
    return Frag(clr, depth);
}

};
```

FILE **scatter.cpp**

```
#include <pt/material/scatter.h>

namespace pt
{

//-----------------------------------------------------------------------------
Scatter::Scatter(Color clr, double k_coef, double s_coef) : clr(clr), k_coef(k_coef), s_coef(s_coef)
↪  {}

//-----------------------------------------------------------------------------
ScatterType Scatter::scatter(const Ray& ray,
                     const Intersection& inter,
                     Color& clrAbsorbtion,
                     Ray& scattered,
                     double& diffusion) const {
    clrAbsorbtion = clr*(k_coef + s_coef*dot(vec3(-ray.dir).normalize(),
    ↪  vec3(inter.normal).normalize()));
    scattered.pos = inter.pos;
    scattered.dir = inter.normal;
    diffusion = 1;
    return SCATTER_RAYTRACING_END;
}

};
```

FILE **scene.cpp**

```
#include <iostream>

#include <pt/object/scene.h>
#include <pt/shape/polygon.h>
#include <pt/object/texture_polygon.h>
#include <pt/material/scatter.h>
#include <pt/shape/portals.h>
```

```cpp
#include <pt/pt2easybmp.h>
#include <stb_image.h>

namespace pt
{

//-----------------------------------------------------------------------------
bool Scene::intersect(const Ray& ray,
                      Intersection& inter,
                      double tMin,
                      double tMax) const {
    if (array.size() > 0) {
        Intersection inter1;
        inter.t = tMax;
        bool isIntersect = array[0]->intersect(ray, inter, tMin, tMax);;
        inter.data.type = 0;
        for (int i = 1; i < array.size(); ++i) {
            if (array[i]->intersect(ray, inter1, tMin, tMax)) {
                isIntersect = true;
                if (inter1.t <= inter.t) {
                    inter = inter1;
                    inter.data.type = i;
                }
            }
        }
        return isIntersect;
    } else
        return false;
}

//-----------------------------------------------------------------------------
ScatterType Scene::scatter(const Ray& ray,
                    const Intersection& inter,
                    Color& clrAbsorbtion,
                    Ray& scattered,
                    double& diffusion) const {
    return array[inter.data.type]->scatter(ray, inter, clrAbsorbtion, scattered, diffusion);
}

//-----------------------------------------------------------------------------
void Scene::add(Object_ptr obj) {
    array.push_back(obj);
}

//-----------------------------------------------------------------------------
Scene loadFrame(const scene::Frame& frame) {
    Scene result;
    // Считываем раскрашенные полигоны
    for (auto& i : frame.colored_polygons) {
        result.array.push_back(makePolygon(
            i.polygon,
            i.crd,
            makeScatter(Color(i.color.x, i.color.y, i.color.z))
        ));
    }

    // Считываем порталы
    for (auto& i : frame.portals) {
        result.array.push_back(makePortals(
            i.crd2, i.crd1,
            i.polygon,
            makeScatter(Color(i.color1.x, i.color1.y, i.color1.z)),
            makeScatter(Color(i.color2.x, i.color2.y, i.color2.z))
        ));
    }

    // Считываем текстуры
    std::vector<Image_ptr> images;
    for (auto& i : frame.textures) {
        Image_ptr img(new Image(1, 1));
        loadAsPng(*img, i.filename);
        images.push_back(img);
    }

    // Считываем текстурированные полигоны
```

```
    for (auto& i : frame.textured_polygons) {
        space2 line1, line2, newSpace;

        // Подбираем такие 3 точки, чтобы получилась невырожденная система координат
        for (int j = 0; j < i.polygon.size() - 2; j++) {
            auto a = i.polygon[j];
            auto b = i.polygon[j+1];
            auto c = i.polygon[j+2];

            auto at = i.tex_coords[j];
            auto bt = i.tex_coords[j+1];
            auto ct = i.tex_coords[j+2];

            line1 = space2(b - a, c - a, a);
            line2 = space2(bt - at, ct - at, at);

            newSpace = combine(invert(line2), line1);

            // Проверяем на невырожденность
            for (int k = 0; k < i.polygon.size(); k++) {
                if (!isNear(i.tex_coords[k], newSpace.to(i.polygon[k]))) {
                    goto next_iteration;
                }
            }

            // Если проверка не разу не случилась, то нам подходит эта система координат
            goto end_cycle;

            next_iteration:;
        }

        // В нормальном случае мы должны перескочить этот участок
        std::cout << "You have line figure with texture. That's bad." << std::endl;

        end_cycle:
        result.array.push_back(makeTexturePolygon(
            i.polygon,
            i.crd,
            images[i.texture_id],
            newSpace
        ));
    }

    return result;
}

};
```

FILE **sky.cpp**

```
#include <pt/object/sky.h>

namespace pt
{

//---------------------------------------------------------------------------
Sky::Sky(Color clr1, Color clr2) : clr1(clr1), clr2(clr2) {}

//---------------------------------------------------------------------------
bool Sky::intersect(const Ray& ray,
                    Intersection& inter,
                    double tMin,
                    double tMax) const {
    inter.t = tMax;
    inter.pos = ray.pos + ray.dir * inter.t;
    inter.normal = ray.dir;
    return true;
}

//---------------------------------------------------------------------------
ScatterType Sky::scatter(const Ray& ray,
                         const Intersection& inter,
                         Color& clrAbsorbtion,
                         Ray& scattered,
                         double& diffusion) const {
```

```
    vec3 dir = ray.dir;
    double k = (dir.normalize().z + 1) * 0.5;
    clrAbsorbtion.r = clr1.r * (1 - k) + clr2.r * k;
    clrAbsorbtion.g = clr1.g * (1 - k) + clr2.g * k;
    clrAbsorbtion.b = clr1.b * (1 - k) + clr2.b * k;
    clrAbsorbtion.a = 1;
    diffusion = 0;
    return SCATTER_END;
}

};
```

FILE **sphere.cpp**

```
#include <pt/shape/sphere.h>

namespace pt
{

//---------------------------------------------------------------------------
bool Sphere::intersect(const Ray& ray,
                       Intersection& inter,
                       double tMin,
                       double tMax) const {
    vec3 V = ray.dir;
    vec3 P = ray.pos - A;

    double vv = dot(V, V);
    double pp = dot(P, P);
    double pv = dot(P, V);

    double a = vv;
    double b = 2.0 * pv;
    double c = pp - r*r;

    double d = b*b - 4.0*a*c;
    if (d >= 0) {
        d = sqrt(d);
        double t1 = (-b-d)/(2.0*a);
        double t2 = (-b+d)/(2.0*a);
        if (t1 > tMin && t1 < tMax) {
            inter.t = t1;
            inter.pos = ray.pos + ray.dir * t1;
            inter.normal = inter.pos - A;
            inter.normal.normalize();
            //if (dot(ray.dir, inter.normal) >= 0)
                //inter.normal = -inter.normal;
            return true;
        } else
        if (t2 > tMin && t2 < tMax) {
            inter.t = t2;
            inter.pos = ray.pos + ray.dir * t2;
            inter.normal = inter.pos - A;
            inter.normal.normalize();
            //if (dot(ray.dir, inter.normal) >= 0)
                //inter.normal = -inter.normal;
            return true;
        } else
            return false;
    } else
        return false;
}

};
```

FILE **texture_polygon.cpp**

```
#include <pt/object/texture_polygon.h>

namespace pt
{

//---------------------------------------------------------------------------
```

```
TexturePolygon::TexturePolygon(const std::vector<vec2>& polygon, crd3 coords, Image_ptr img, const space2& tr, double k_coef, double s_coef) : img(img), array(polygon), coords(coords), tr(tr), k_coef(k_coef), s_coef(s_coef) {
    d = -dot(coords.k, coords.pos);
    normal = coords.k;

    min.x = +std::numeric_limits<double>::infinity();
    min.y = +std::numeric_limits<double>::infinity();
    max.x = -std::numeric_limits<double>::infinity();
    max.y = -std::numeric_limits<double>::infinity();

    for (size_t i = 0; i < polygon.size(); i++) {
        if (polygon[i].x < min.x) min.x = polygon[i].x;
        if (polygon[i].y < min.y) min.y = polygon[i].y;
        if (polygon[i].x > max.x) max.x = polygon[i].x;
        if (polygon[i].y > max.y) max.y = polygon[i].y;
    }
}

//-----------------------------------------------------------------------------
bool TexturePolygon::intersect(const Ray& ray,
                    Intersection& inter,
                    double tMin,
                    double tMax) const {
    if (dot(ray.dir, normal) != 0) {
        double t = (-d - dot(ray.pos, normal))/dot(ray.dir, normal);

        if (t > tMin && t < tMax) {
            // Position when ray intersect plane
            vec3 x = ray.pos + ray.dir * t;

            vec2 r;
            r.x = (dot(x, coords.i) - dot(coords.pos, coords.i))/dot(coords.i, coords.i);
            r.y = (dot(x, coords.j) - dot(coords.pos, coords.j))/dot(coords.j, coords.j);

            bool inRect = r.x >= min.x && r.y >= min.y && r.x <= max.x && r.y <= max.y;
            if (!inRect)
                return false;

            vec2 imgPos = tr.to(r);

            // Преобразование координат из [0, 1]x[0, 1] в координаты изображения. Так же текстура делается повторяющейся.
            imgPos.x += int(std::fabs(imgPos.x) + 2);
            imgPos.y += int(std::fabs(imgPos.y) + 2);
            imgPos.x = std::fmodf(imgPos.x, 1);
            imgPos.y = std::fmodf(imgPos.y, 1);
            imgPos.x *= img->getWidth();
            imgPos.y *= img->getHeight();


            if (imgPos.x < 0 || imgPos.x > img->getWidth() ||
                imgPos.y < 0 || imgPos.y > img->getHeight())
                return false;
            if (img->operator()(imgPos.x, imgPos.y).a == 0)
                return false;

            if (pointInPolygon(array, r)) {
                inter.data.vector = imgPos;
                inter.t = t;
                if (dot(ray.dir, normal) < 0)
                    inter.normal = normal;
                else
                    inter.normal = -normal;
                inter.pos = x;
                return true;
            } else
                return false;
        } else
            // Ray is not in ranges
            return false;
    } else {
        // Ray is parallel plane
        return false;
    }
```

```
}

//-----------------------------------------------------------------------------
ScatterType TexturePolygon::scatter(const Ray& ray,
                        const Intersection& inter,
                        Color& clrAbsorbtion,
                        Ray& scattered,
                        double& diffusion) const {
    Scatter s(img->operator()(inter.data.vector.x, inter.data.vector.y), k_coef, s_coef);
    return s.scatter(ray, inter, clrAbsorbtion, scattered, diffusion);
}

};
```

FILE **triangle.cpp**

```
#include <pt/shape/triangle.h>

namespace pt
{

//-----------------------------------------------------------------------------
double area(double a, double b, double c) {
    double p = (a + b + c)/2.0f;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
//-----------------------------------------------------------------------------
Triangle::Triangle(vec3 a, vec3 b, vec3 c, Material_ptr material) :
    a(a), b(b), c(c),
    Shape(material) {
    // Equation of plane, every x in plane when: (n, x) + d = 0
    normal = (cross(b-a, c-b)).normalize();
    d = -dot(normal, c);
    ab = distance(b, a);
    bc = distance(c, b);
    ac = distance(c, a);
    S = area(ab, bc, ac);
}

//-----------------------------------------------------------------------------
bool Triangle::intersect(const Ray& ray,
                        Intersection& inter,
                        double tMin,
                        double tMax) const {
    if (dot(ray.dir, normal)) {
        double t = (-d - dot(ray.pos, normal))/dot(ray.dir, normal);

        if (t > tMin && t < tMax) {
            // Position when ray intersect plane
            vec3 x = ray.pos + ray.dir * t;

            // Point in triangle <=> area of triangle = sum of inner triangles
            double xa = distance(a, x);
            double xb = distance(b, x);
            double xc = distance(c, x);

            double S1 = area(ab, xa, xb);
            double S2 = area(bc, xb, xc);
            double S3 = area(ac, xa, xc);

            if (fabs((S1 + S2 + S3)/S) < 1.0001f) {
                inter.t = t;
                if (dot(ray.dir, normal) < 0)
                    inter.normal = normal;
                else
                    inter.normal = -normal;
                inter.pos = x;
                return true;
            } else
                return false;
        } else
            // Ray is not in ranges
            return false;
    } else {
        // Ray is parallel plane
```

```
        return false;
    }
}

};
```

5.3 Сцены

FILE **hello_world.cpp**

```
#include <pt/shape/triangle.h>

namespace pt
{

//---------------------------------------------------------------------------
double area(double a, double b, double c) {
    double p = (a + b + c)/2.0f;
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
//---------------------------------------------------------------------------
Triangle::Triangle(vec3 a, vec3 b, vec3 c, Material_ptr material) :
    a(a), b(b), c(c),
    Shape(material) {
    // Equation of plane, every x in plane when: (n, x) + d = 0
    normal = (cross(b-a, c-b)).normalize();
    d = -dot(normal, c);
    ab = distance(b, a);
    bc = distance(c, b);
    ac = distance(c, a);
    S = area(ab, bc, ac);
}

//---------------------------------------------------------------------------
bool Triangle::intersect(const Ray& ray,
                         Intersection& inter,
                         double tMin,
                         double tMax) const {
    if (dot(ray.dir, normal)) {
        double t = (-d - dot(ray.pos, normal))/dot(ray.dir, normal);

        if (t > tMin && t < tMax) {
            // Position when ray intersect plane
            vec3 x = ray.pos + ray.dir * t;

            // Point in triangle <=> area of triangle = sum of inner triangles
            double xa = distance(a, x);
            double xb = distance(b, x);
            double xc = distance(c, x);

            double S1 = area(ab, xa, xb);
            double S2 = area(bc, xb, xc);
            double S3 = area(ac, xa, xc);

            if (fabs((S1 + S2 + S3)/S) < 1.0001f) {
                inter.t = t;
                if (dot(ray.dir, normal) < 0)
                    inter.normal = normal;
                else
                    inter.normal = -normal;
                inter.pos = x;
                return true;
            } else
                return false;
        } else
            // Ray is not in ranges
            return false;
    } else {
        // Ray is parallel plane
        return false;
    }
}
```

```
};
```

FILE **polygon_test.cpp**

```cpp
#include <pt/pt.h>
#include <pt/object/scene.h>
#include <pt/object/sky.h>
#include <pt/shape/triangle.h>
#include <pt/shape/polygon.h>
#include <pt/shape/sphere.h>
#include <pt/material/scatter.h>
#include <pt/renderer.h>
#include <pt/pt2easybmp.h>

using namespace pt;

void initScene(Scene& scene) {
    auto& array = scene.array;
    Material_ptr sc = makeScatter(Color(0.5, 0.5, 0.9));

    {
        const double size = 500;
        const double depth = -1.5;
        vec3 a(-size, -size, depth);
        vec3 b(-size, size, depth);
        vec3 c(size, size, depth);
        vec3 d(size, -size, depth);
        scene.add(makeTriangle(a, b, c, makeScatter(pt::Color(0.6, 0.6, 0.6))));
        scene.add(makeTriangle(c, d, a, makeScatter(pt::Color(0.6, 0.6, 0.6))));
    }

    crd3 coords1;
    coords1.pos = vec3(0);
    coords1.i = vec3(1, 0, 0);
    coords1.j = vec3(0, 0, 1);
    coords1.k = vec3(0, 1, 0);

    {
        std::vector<vec2> mas;
        mas.push_back({0, 0});
        mas.push_back({0, 1});
        mas.push_back({1, 0});
        array.push_back(makePolygon(mas, coords1, sc));

        crd3 coords2;
        coords2.pos = vec3(0, 2, -0.5);
        coords2.i = vec3(1.0/sqrt(2.0), 0, -1.0/sqrt(2.0));
        coords2.j = vec3(1.0/sqrt(2.0), 0, 1.0/sqrt(2.0));
        coords2.k = cross(coords2.i, coords2.j);
        coords2.i.normalize();
        coords2.j.normalize();
        coords2.k.normalize();
        array.push_back(makePolygon(mas, coords2, sc));

        coords2.pos = vec3(1.5, 2, -0.5);
        coords2.i = vec3(1.0/sqrt(2.0), -1.0/sqrt(2.0), 0);
        coords2.j = vec3(1.0/sqrt(2.0), 0, 1.0/sqrt(2.0));
        coords2.k = cross(coords2.i, coords2.j);
        coords2.i.normalize();
        coords2.j.normalize();
        coords2.k.normalize();
        array.push_back(makePolygon(mas, coords2, sc));

        coords2.pos = vec3(-1.5, 2, -0.5);
        coords2.i = vec3(1.0/sqrt(2.0), -1.0/sqrt(2.0), 0);
        coords2.j = vec3(1.0/sqrt(2.0), 1.0/sqrt(2.0), 0);
        coords2.k = cross(coords2.i, coords2.j);
        coords2.i.normalize();
        coords2.j.normalize();
        coords2.k.normalize();
        array.push_back(makePolygon(mas, coords2, sc));
    }

    {
```

```
        crd3 coords2 = coords1;
        coords2.pos = vec3(1.5, 0, 0);
        std::vector<vec2> mas;
        mas.push_back({0, 0});
        mas.push_back({0, 1});
        mas.push_back({1, 1});
        mas.push_back({1, 0});
        array.push_back(makePolygon(mas, coords2, sc));
    }

    {
        crd3 coords2 = coords1;
        coords2.pos = vec3(-1.5, 0, 0);
        std::vector<vec2> mas;
        for (int i = 0; i < 359; i++)
            mas.push_back({0.5 + 0.5 * sin(i/180.0 * pi), 0.5 + 0.5 * cos(i/180.0 * pi)});
        array.push_back(makePolygon(mas, coords2, sc));
    }

    {
        crd3 coords2 = coords1;
        coords2.pos = vec3(-3, 0, 0);
        std::vector<vec2> mas;
        mas.push_back({0, 0});
        mas.push_back({0, 1});
        mas.push_back({1, 0});
        mas.push_back({1, 1});
        array.push_back(makePolygon(mas, coords2, sc));
    }

    {
        crd3 coords2 = coords1;
        coords2.pos = vec3(3, 0, 0);
        std::vector<vec2> mas;
        mas.push_back({31/290.0, 34/313.0});
        mas.push_back({290/290.0, 100/313.0});
        mas.push_back({0/290.0, 199/313.0});
        mas.push_back({165/290.0, 313/313.0});
        mas.push_back({271/290.0, 226/313.0});
        mas.push_back({177/290.0, 0/313.0});
        mas.push_back({167/290.0, 240/313.0});
        mas.push_back({55/290.0, 108/313.0});
        mas.push_back({194/290.0, 113/313.0});
        array.push_back(makePolygon(mas, coords2, sc));
    }

    scene.add(makeSky(Color(1, 1, 1), Color(0, 0.2, 0.2)));
}

int main() {
    Scene scene;
    initScene(scene);

    Image img(1000, 500);

    PerspectiveCamera cam(2, 45*pt::pi/180, 0, vec3(0, 7, 1), img.getWidth(), img.getHeight());
    cam.lookAt(vec3(0, 0, 0));

    RayTracing ren2(2, 4, true);
    ren2.setAmbientLight(Color(0.5, 0.5, 0.5, 1));
    ren2.luminaries.push_back(PointLight(vec3(0, 0, 5), pt::Color(0.5, 0.5, 0.5).sqrt()));
    ren2.luminaries.push_back(PointLight(vec3(5, 5, 3), pt::Color(0.5, 0.5, 0.5).sqrt()));
    ren2.luminaries.push_back(PointLight(vec3(-5, -3, 3), pt::Color(0.5, 0.5, 0.5).sqrt()));
    ren2.assign(&cam, &scene, &img);
    ren2.render();
    img.colorCorrection();
    saveAsPng(img, ”polygon_test.png”);

    return 0;
}
```

FILE **portal_light_test.cpp**

```cpp
#include <iostream>
#include <windows.h>

#include <pt/pt.h>
#include <pt/object/scene.h>
#include <pt/object/sky.h>
#include <pt/shape/sphere.h>
#include <pt/shape/polygon.h>
#include <pt/shape/portals.h>
#include <pt/material/scatter.h>
#include <pt/material/reflect.h>
#include <pt/camera/360.h>
#include <pt/camera/orthogonal.h>
#include <pt/renderer.h>
#include <pt/pt2easybmp.h>

int main() {
    using namespace pt;
    bool isDrawHints = false;

    Image img(1000, 500);
    Scene scene;

    // Создаем камеру
    vec3 camPos = {0, -4, 4};
    vec3 lookAt = {0, 0, 0.5};
    PerspectiveCamera cam(1, pi / 2.0, 0, camPos, img.getWidth(), img.getHeight());
    cam.lookAt(lookAt);

    RayTracing ren(2, 4, true);
    ren.setAmbientLight(Color(0.7, 0.7, 0.7, 1));

    // Создаем пол
    const double size = 500;
    std::vector<vec2> mas2;
    mas2.push_back({-size, -size});
    mas2.push_back({-size, size});
    mas2.push_back({size, size});
    mas2.push_back({size, -size});
    crd3 floor = getStandardCrd3();
    scene.add(makePolygon(mas2, floor, makeScatter(Color(0.4, 0, 0.6))));

    Color portalFirstColor0 = Color(1, 0.5, 0.15); // orange
    Color portalSecondColor0 = Color(0.1, 0.55, 1); // blue
    Color portalFirstColor1 = Color(0.67, 0.02, 0.02); // red
    Color portalSecondColor1 = Color(0.33, 0, 1); // dark blue

    // Массивы для порталов
    std::vector<vec2> mas;
    mas.push_back({-1, -1});
    mas.push_back({-1, 1});
    mas.push_back({1, 1});
    mas.push_back({1, -1});

    double h = 0.1;
    std::vector<vec2> contour;
    contour.push_back({-1, 1});
    contour.push_back({-1-h, 1+h});
    contour.push_back({1+h, 1+h});
    contour.push_back({1+h, -1-h});
    contour.push_back({-1-h, -1-h});
    contour.push_back({-1-h, 1+h});
    contour.push_back({-1, 1});
    contour.push_back({1, 1});
    contour.push_back({1, -1});
    contour.push_back({-1, -1});

    // Первая пара порталов
    crd3 p11 = getStandardCrd3();
    crd3 p12 = getStandardCrd3();
    p11.pos = {0, 0, 1};
    p12.pos = {2 + 2*h, 2, 1};
    Portals_ptr prt = makePortals(p11, p12, mas, makeScatter(portalFirstColor0),
    ↪  makeScatter(portalSecondColor0));
```

```
    scene.add(prt);
    ren.addPortal(prt);
    scene.add(makePolygon(contour, p11, makeScatter(portalFirstColor0)));
    scene.add(makePolygon(contour, p12, makeScatter(portalSecondColor0)));

    // Вторая пара порталов
    crd3 p21 = getStandardCrd3();
    crd3 p22 = getStandardCrd3();
    p21.pos = p12.pos;
    p21.pos.z += 1;
    p22.pos = {0, 2, 2};
    Portals_ptr prt1 = makePortals(p21, p22, mas, makeScatter(portalFirstColor1),
    ↪  makeScatter(portalSecondColor1));
    scene.add(prt1);
    ren.addPortal(prt1);
    scene.add(makePolygon(contour, p21, makeScatter(portalFirstColor1)));
    scene.add(makePolygon(contour, p22, makeScatter(portalSecondColor1)));

    // Источник освещения
    vec3 lightPos = p22.pos;
    lightPos.z += 1;
    lightPos.x += 0.5;
    ren.luminaries.push_back(PointLight(lightPos, Color(0.5, 0.5, 0.5)));

    // Добавляем сферы, которые показывают положение источника освещения
    double spsize = 0.05;
    vec3 spherePos = lightPos + vec3(0, 0, spsize + 0.01);
    scene.add(makeSphere(spherePos, spsize, makeScatter(Color(1, 1, 1, 1))));

    if (isDrawHints) {
        vec3 spherePos1 = space3(prt1->p1).from(space3(prt1->p2).to(spherePos));
        scene.add(makeSphere(spherePos1, spsize, makeScatter(Color(0.5, 0.5, 0.5, 1))));
        vec3 spherePos2 = space3(prt->p1).from(space3(prt->p2).to(spherePos1));
        scene.add(makeSphere(spherePos2, spsize, makeScatter(Color(0.25, 0.25, 0.25, 1))));
    }

    // Добавляем небо
    scene.add(makeSky(Color(0.3, 0.3, 0.9), Color(1, 1, 1)));

    // Для отладки
    /*Ray ray;
    ray.pos = vec3(0, -4, 1);
    ray.dir = -ray.pos.normalize();
    ren.computeColor(ray);*/

    // Рендерим первую картинку
    ren.assign(&cam, &scene, &img);
    ren.render();
    img.colorCorrection();
    if (isDrawHints)
        saveAsPng(img, "portal_light_test1_hints_1.png");
    else
        saveAsPng(img, "portal_light_test1_1.png");

    // Рендерим вторую картинку с другого угла
    cam.pos = {0, -3, 1.2};
    cam.lookAt({0, 0, 0.5});
    ren.render();
    img.colorCorrection();
    if (isDrawHints)
        saveAsPng(img, "portal_light_test1_hints_2.png");
    else
        saveAsPng(img, "portal_light_test1_2.png");

    // Рендерим картинку сверху
    Orthogonal cam2(vec3(1, 1, 5), 10.0 / img.getWidth() , img.getWidth(), img.getHeight());
    cam2.lookTowards(vec3(1, 1 - 0.001, 0));
    ren.assign(&cam2, &scene, &img);
    ren.render();
    img.colorCorrection();
    if (isDrawHints)
        saveAsPng(img, "portal_light_test1_hints_3.png");
    else
        saveAsPng(img, "portal_light_test1_3.png");
}
```

FILE **scene_raytracing.cpp**

```cpp
#include <fstream>
#include <iostream>
#include <iomanip>

#include <pt/pt.h>
#include <pt/object/scene.h>
#include <pt/object/sky.h>
#include <pt/shape/sphere.h>
#include <pt/shape/triangle.h>
#include <pt/shape/cone.h>
#include <pt/material/scatter.h>
#include <pt/material/reflect.h>
#include <pt/material/light.h>
#include <pt/camera/360.h>
#include <pt/camera/orthogonal.h>
#include <pt/renderer.h>
#include <pt/pt2easybmp.h>

#include <prtl_vis/scene_reader.h>

int main(int argc, char** argv) {
    std::string settingsFile = "settings.json";
    std::string filename = "scene.json";
    if (argc > 1) {
        filename = std::string(argv[1]);
        std::cout << "Read file `" << filename << "`" << std::endl;
    } else {
        std::cout << "Please specify file to be rendered in command line arguments." << std::endl;
        std::cout << "Filename interpreted as `" << filename << "`." << std::endl;
    }

    scene::json js;
    try {
        std::ifstream fin(filename);
        fin >> js;
        fin.close();
    } catch (...) {
        std::cout << "File `scene.json` didn't exists or it not contains the scene." << std::endl;
        std::cout << "Terminate program." << std::endl;
        system("pause");
        return 1;
    }

    auto scenejs = scene::parseScene(js);
    spob::vec3 lookAt = scenejs.cam_rotate_around;
    spob::vec3 pos = spheric2cartesian(scenejs.cam_spheric_pos) + lookAt;

    int width, height;
    bool isUsePathTracing;
    bool isDrawDepth;
    int rayTracingSamples;
    int pathTracingSamples;
    int threads;
    bool isLog;

    scene::json settings;
    try {
        std::ifstream fin(settingsFile);
        fin >> settings;
        fin.close();
    } catch (...) {
        std::cout << "Settings file is clear or didn't exists." << std::endl;
        std::cout << "Created standard `settings.json`." << std::endl;
        settings = scene::json();
        settings["width"] = 1000;
        settings["height"] = int(settings["width"]) / 2;
        settings["isUsePathTracing"] = false;
        settings["isDrawDepth"] = false;
        settings["rayTracingSamples"] = 2;
        settings["pathTracingSamples"] = 200;
        settings["threads"] = 4;
        settings["isLog"] = true;
        std::ofstream fout(settingsFile);
        fout << std::setw(4) << settings;
```

```
        fout.close();

        system("pause");
    }

    width = settings["width"];
    height = settings["height"];
    isUsePathTracing = settings["isUsePathTracing"];
    isDrawDepth = settings["isDrawDepth"];
    rayTracingSamples = settings["rayTracingSamples"];
    pathTracingSamples = settings["pathTracingSamples"];
    threads = settings["threads"];
    isLog = settings["isLog"];

    std::cout << std::endl << std::endl;

    using namespace pt;

    for (int i = 0; i < scenejs.frames.size(); i++) {
        if (isLog)
            std::cout << "Rendering " << i << " frame of " << scenejs.frames.size() << " frames
            ↪  total" << std::endl;;
        Image img(width, height);
        Image dImg(width, height);
        Scene scene = loadFrame(scenejs.frames[i]);
        scene.add(makeSky(Color(0.3, 0.3, 0.9), Color(1, 1, 1)));

        PerspectiveCamera cam(1, pi/2.0, 0, pos, img.getWidth(), img.getHeight());
        cam.lookAt(lookAt);
        StandardRenderer* ren;
        if (isUsePathTracing)
            ren = new PathTracing(pathTracingSamples, threads, isLog, 30);
        else
            ren = new RayTracing(rayTracingSamples, threads, isLog, 100);
        ren->setAmbientLight(Color(1, 1, 1, 1));
        ren->luminaries.push_back(PointLight(vec3(0, 0, 1), Color(1.5, 1.5, 1.5)));
        ren->luminaries.push_back(PointLight(vec3(0, 1, 2.9), Color(0.5, 0.5, 0.5)));
        ren->assign(&cam, &scene, &img, &dImg);
        ren->render();

        img.colorCorrection();
        saveAsPng(img, filename + "_" + std::to_string(i) + ".png");

        if (isDrawDepth) {
            toGrayScaleDoubleImg(dImg);
            saveAsPng(dImg, filename + "_" + std::to_string(i) + "_depth.png");
            saveAsDoubleImg(&img, filename + "_" + std::to_string(i) + ".double");
        }

        delete ren;
    }
}
```

FILE **standard_scene_2.cpp**

```
#include <iostream>
#include <random>

#include <pt/camera/360.h>
#include <pt/camera/orthogonal.h>
#include <pt/material/reflect.h>
#include <pt/material/refract.h>
#include <pt/material/scatter.h>
#include <pt/material/light.h>
#include <pt/object/scene.h>
#include <pt/object/sky.h>
#include <pt/pt.h>
#include <pt/pt2easybmp.h>
#include <pt/renderer.h>
#include <pt/shape/sphere.h>
#include <pt/shape/triangle.h>

using namespace pt;

//---------------------------------------------------------------------------
```

```
float myrandom() {
    static std::mt19937 generator(1);
    static std::uniform_real_distribution<float> distribution(0, 1);
    return distribution(generator);
}

//-----------------------------------------------------------------------------
void rect(Scene& scene, vec3 a, vec3 b, vec3 c, Material_ptr m) {
    std::swap(a.y, a.z);
    std::swap(b.y, b.z);
    std::swap(c.y, c.z);
    //std::vector<vec2> mas = {a, b, c, b - (b-(c+a)*0.5)*2};
    //scene.add(makeTriangle(a, b, c, m));
    //scene.add(makeTriangle(a, b - (b-(c+a)*0.5)*2, c, m));
    crd3 crd = spob::makePlane3(b, a, c);
    double il = crd.i.length();
    double jl = crd.j.length();
    std::vector<vec2> mas = {{0, 0}, {0, jl}, {il, jl}, {il, 0}};
    crd.i.normalize();
    crd.j.normalize();
    crd.k.normalize();
    scene.add(makePolygon(mas, crd, m));
}

//-----------------------------------------------------------------------------
void makeScene1(Scene& scene) {
    double k_coef = 1;
    double s_coef = 0;

    // FLoor
    rect(scene,
        vec3(-2, 0, -2),
        vec3(-2, 0, 2),
        vec3(20, 0, 2),
        makeScatter(Color(0.7f, 0.7f, 0.7f), k_coef, s_coef)
    );

    // Ceiling
    rect(scene,
        vec3(-2, 4, -2),
        vec3(-2, 4, 2),
        vec3(20, 4, 2),
        makeScatter(Color(0.9f, 0.9f, 0.9f), k_coef, s_coef)
    );

    // Front wall
    rect(scene,
        vec3(-2, 0, -2),
        vec3(-2, 0, 2),
        vec3(-2, 4, 2),
        makeScatter(Color(0.7f, 0.7f, 0.7f), k_coef, s_coef)
    );

    // Left wall
    rect(scene,
        vec3(-2, 0, -2),
        vec3(20, 0, -2),
        vec3(20, 4, -2),
        makeScatter(Color(0.9f, 0.1f, 0.1f), k_coef, s_coef)
    );

    // Right wall
    rect(scene,
        vec3(-2, 0, 2),
        vec3(20, 0, 2),
        vec3(20, 4, 2),
        makeScatter(Color(0.1f, 0.9f, 0.1f), k_coef, s_coef)
    );

    scene.add(makeSphere(
        vec3(-1, 1, 1), 1,
        makeReflect(Color(1, 1, 1), 0)
    ));

```

```
    scene.add(makeSphere(vec3(-1.3, -1.3, 0.7), 0.7, makeScatter(Color(0.1, 0.1, 0.9), k_coef,
    ↪  s_coef)));
    scene.add(makeSphere(vec3(1, 0.2, 0.5), 0.5, makeRefract(1.5, 0)));

    float lampSize = 1;
    rect(scene,
        vec3(-lampSize, 3.95, -lampSize),
        vec3(-lampSize, 3.95, lampSize),
        vec3(lampSize, 3.95, lampSize),
        makeLight(Color(1, 1, 1))
    );

    scene.add(makeSky(Color(0, 0, 0), Color(0, 0, 0)));
}

//-----------------------------------------------------------------------------
void makeCam1(PerspectiveCamera& cam1, Orthogonal& cam2, Camera360& cam3, const Image& img, const
↪  Image& img2) {
    vec3 lookAt(0, 0, 2);
    vec3 pos(15, 0, 2);
    vec3 sppos = spob::cartesian2spheric(pos - lookAt);

    // Инициализируем камеру с песпективной проекцией
    double beta = pi/6;
    double alpha = pi;
    cam1 = PerspectiveCamera(12, 20*pi/180.0, 0.1, lookAt + spheric2cartesian(vec3(sppos.x + 0 *
    ↪  pi/180, sppos.y, sppos.z)), img.getWidth(), img.getHeight());
    cam1.lookAt(lookAt);

    // Инициализируем камеру с ортогональной проекцией
    cam2 = Orthogonal(pos, 0.01 * img.getHeight() / 500, img.getWidth(), img.getHeight());
    cam2.lookTowards(lookAt);

    // Инициализируем камеру с проекцией на сферу
    cam3 = Camera360(0.18*(pos - lookAt) + lookAt, img2.getHeight());
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------

int main() {
    Image img(500, 500);
    Image img2(4000, 2000);

    Scene scene;
    makeScene1(scene);
    PerspectiveCamera cam1(1, pi / 2.0, 0, vec3(0), img.getWidth(), img.getHeight());
    Orthogonal cam2(vec3(0), 0.006, img.getWidth(), img.getHeight());
    Camera360 cam3(vec3(0), img.getHeight());
    makeCam1(cam1, cam2, cam3, img, img2);

    {
        RayTracing ren(4, 4);
        ren.luminaries.push_back(PointLight(vec3(0, 0, 3.94), Color(1, 1, 1)));
        ren.setAmbientLight(Color(0.5, 0.5, 0.5, 1));

        ren.assign(&cam1, &scene, &img);
        //ren.render();
        img.colorCorrection();
        //saveAsPng(img, "standard_scene_2_ray_perspective.png");

        ren.assign(&cam2, &scene, &img);
        //ren.render();
        img.colorCorrection();
        //saveAsPng(img, "standard_scene_2_ray_orthogonal.png");

        ren.assign(&cam3, &scene, &img2);
        ren.render();
        img2.colorCorrection();
        saveAsPng(img2, "standard_scene_2_ray_360.png");
    }

    {
        PathTracing ren(1000, 3, true, 5);
```

```
        ren.luminaries.clear();
        ren.assign(&cam1, &scene, &img);
        //ren.render();
        img.colorCorrection();
        //saveAsPng(img, "standard_scene_2_path_perspective.png");
    }
}
```

FILE **standard_scene.cpp**

```cpp
#include <iostream>
#include <random>


#include <pt/camera/360.h>
#include <pt/camera/orthogonal.h>
#include <pt/material/reflect.h>
#include <pt/material/refract.h>
#include <pt/material/scatter.h>
#include <pt/object/scene.h>
#include <pt/object/sky.h>
#include <pt/pt.h>
#include <pt/pt2easybmp.h>
#include <pt/renderer.h>
#include <pt/shape/sphere.h>
#include <pt/shape/triangle.h>

using namespace pt;

//-----------------------------------------------------------------------------
float myrandom() {
    static std::mt19937 generator(1);
    static std::uniform_real_distribution<float> distribution(0, 1);
    return distribution(generator);
}

//-----------------------------------------------------------------------------
void makeScene1(Scene& scene) {
    double k_coef = 0.8;
    double s_coef = 0.2;
    auto& array = scene.array;
    array.push_back(makeSky(Color(1, 1, 1), Color(0.5, 0.7, 1).sqrt()));

    for(int i = -11; i < 11; ++i) {
        for(int j = -11; j < 11; ++j) {
            vec3 c(i + 0.9 * myrandom(), j + 0.9 * myrandom(), 0.2);
            double p = myrandom();
            if(p < 0.8) {
                array.push_back(makeSphere(c, 0.2, makeScatter(Color(myrandom(), myrandom(),
                ↪  myrandom()).sqrt(), k_coef, s_coef)));
            } else if(p < 0.95) {
                double r = 0.5 * (1 + myrandom());
                double g = 0.5 * (1 + myrandom());
                double b = 0.5 * (1 + myrandom());
                array.push_back(makeSphere(c, 0.2, makeReflect(Color(r, g, b).sqrt(), 0.5 *
                ↪  myrandom())));
            } else {
                array.push_back(makeSphere(c, 0.2, makeRefract(1.5, 0)));
            }
        }
    }
    array.push_back(makeSphere(vec3(0, 0, 1), 1, makeRefract(1.5, 0)));
    array.push_back(makeSphere(vec3(-4, 0, 1), 1, makeScatter(Color(0.4, 0.2, 0.1).sqrt(), k_coef,
    ↪  s_coef)));
    array.push_back(makeSphere(vec3(4, 0, 1), 1, makeReflect(Color(0.7, 0.6, 0.5).sqrt(), 0)));
    array.push_back(makeSphere(vec3(0, 0, -1000), 1000, makeScatter(Color(0.5, 0.5, 0.5).sqrt(),
    ↪  k_coef, s_coef)));
}

//-----------------------------------------------------------------------------
void makeCam1(PerspectiveCamera& cam1, Orthogonal& cam2, Camera360& cam3, const Image& img) {
    vec3 lookAt(0, 0, 1);
    vec3 pos(12, 3, 2);
    vec3 sppos = spob::cartesian2spheric(pos - lookAt);

```

```
    // Инициализируем камеру с песпективной проекцией
    double beta = pi/6;
    double alpha = pi;
    cam1 = PerspectiveCamera(12, 20*pi/180.0, 0.1, lookAt + spheric2cartesian(vec3(sppos.x + 0 *
    ↪  pi/180, sppos.y, sppos.z)), img.getWidth(), img.getHeight());
    cam1.lookAt(lookAt);

    // Инициализируем камеру с ортогональной проекцией
    cam2 = Orthogonal(pos, 0.03 * img.getHeight() / 500.0, img.getWidth(), img.getHeight());
    cam2.lookTowards(lookAt);

    // Инициализируем камеру с проекцией на сферу
    cam3 = Camera360(0.5*(pos - lookAt) + lookAt, img.getHeight());
}

//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------

int main() {
    Image img(500, 250);
    Image img2(500, 250);

    Scene scene;
    makeScene1(scene);
    PerspectiveCamera cam1(1, pi / 2.0, 0, vec3(0), img.getWidth(), img.getHeight());
    Orthogonal cam2(vec3(0), 0.006, img.getWidth(), img.getHeight());
    Camera360 cam3(vec3(0), img.getHeight());
    makeCam1(cam1, cam2, cam3, img);

    {
        RayTracing ren(1, 4);
        ren.luminaries.push_back(PointLight(vec3(0, 0, 3), Color(1.5, 1.5, 1.5)));
        ren.luminaries.push_back(PointLight(vec3(0, 1, 3), Color(0.5, 0.5, 0.5)));
        ren.setAmbientLight(Color(0.5, 0.5, 0.5, 1));

        ren.assign(&cam1, &scene, &img, &img2);
        ren.render();
        img.colorCorrection();
        toGrayScaleDoubleImg(img2, 500);
        saveAsPng(img, "standard_scene_ray_perspective.png");
        saveAsPng(img2, "standard_scene_ray_perspective_depth.png");

        ren.assign(&cam2, &scene, &img);
        ren.render();
        img.colorCorrection();
        saveAsPng(img, "standard_scene_ray_orthogonal.png");

        ren.assign(&cam3, &scene, &img);
        ren.render();
        img.colorCorrection();
        saveAsPng(img, "standard_scene_ray_360.png");
    }

    {
        PathTracing ren(400, 4);
        ren.luminaries.push_back(PointLight(vec3(0, 0, 3), Color(1.5, 1.5, 1.5)));
        ren.luminaries.push_back(PointLight(vec3(0, 1, 3), Color(0.5, 0.5, 0.5)));
        ren.assign(&cam1, &scene, &img);
        ren.render();
        img.colorCorrection();
        saveAsPng(img, "standard_scene_path_perspective.png");
    }

    system("pause");
}
```