Path tracing

1.0

Generated by Doxygen 1.10.0

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 AABB Struct Reference

Public Member Functions

- constexpr AABB (const Math::Vector3f &a, const Math::Vector3f &b) noexcept
- constexpr **AABB** (const AABB &other) noexcept=default
- constexpr AABB (const AABB &a, const AABB &b) noexcept
- constexpr float GetSurfaceArea () const noexcept
- constexpr bool IntersectsRay (const Ray &ray, float t_min, float t_max) const noexcept

Static Public Member Functions

- static constexpr AABB Empty () noexcept

Public Attributes

- Math::Vector3f min
- Math::Vector3f max

3.1.1 Detailed Description

Definition at line 6 of file AABB.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AABB() [1/2]

```
constexpr AABB::AABB (
            const Math::Vector3f & a,
            const Math::Vector3f & b )  [inline], [constexpr], [noexcept]
```

Definition at line 11 of file AABB.h.

```
00011                                                                      :
00012         min(Math::Min(a, b)), max(Math::Max(a, b)) {}
```

3.1.2.2 AABB() [2/2]

```
constexpr AABB::AABB (
            const AABB & a,
            const AABB & b )  [inline], [constexpr], [noexcept]
```

Definition at line 16 of file AABB.h.

```
                                                          :
        min(Math::Min(a.min, b.min)), max(Math::Max(a.max, b.max)) {}
```

3.1.3 Member Function Documentation

3.1.3.1 Empty()

```
static constexpr AABB AABB::Empty ( )  [inline], [static], [constexpr], [noexcept]
```

Definition at line 56 of file AABB.h.

```
                                                {
        AABB aabb;
        aabb.min = Math::Vector3f(+Math::Constants::Infinity<float>);
        aabb.max = Math::Vector3f(-Math::Constants::Infinity<float>);

        return aabb;
    }
```

3.1.3.2 GetSurfaceArea()

```
constexpr float AABB::GetSurfaceArea ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 19 of file AABB.h.

```
                                                         {
        float a, b, c;
        a = max.x - min.x;
        b = max.y - min.y;
        c = max.z - min.z;
        return 2.f * a * b + 2.f * b * c + 2.f * a * c;
    }
```

3.1.3.3 IntersectsRay()

```
constexpr bool AABB::IntersectsRay (
            const Ray & ray,
            float t_min,
            float t_max ) const  [inline], [constexpr], [noexcept]
```

Definition at line 27 of file AABB.h.

```
                                                                                                   {
        auto one_over_direction = 1.f / ray.direction;
        auto origin = ray.origin;

        auto t0 = (min - origin) * one_over_direction;
        auto t1 = (max - origin) * one_over_direction;

        if (one_over_direction.x < 0.f) {
            std::swap(t0.x, t1.x);
        }

        if (one_over_direction.y < 0.f) {
            std::swap(t0.y, t1.y);
        }

        if (one_over_direction.z < 0.f) {
            std::swap(t0.z, t1.z);
        }

        t_min = Math::Max(t_min, Math::Max(Math::Max(t0.x, t0.y), t0.z));
        t_max = Math::Min(t_max, Math::Min(Math::Min(t1.x, t1.y), t1.z));

        if (t_max <= t_min) {
            return false;
        }

        return true;
    }
```

3.1.4 Member Data Documentation

3.1.4.1 max

```
Math::Vector3f AABB::max
```

Definition at line 7 of file AABB.h.

3.1.4.2 min

```
Math::Vector3f AABB::min
```

Definition at line 7 of file AABB.h.

The documentation for this struct was generated from the following file:

- AABB.h

3.2 AccelerationStructure Class Reference

Public Member Functions

- AccelerationStructure (std::span< HittableObjectPtr > objects) noexcept
- void Update (std::span< HittableObjectPtr > objects) noexcept
- void Hit (const Ray &ray, float tMin, float tMax, HitPayload &payload) const noexcept

3.2.1 Detailed Description

Definition at line 13 of file AccelerationStructure.h.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 AccelerationStructure() [1/2]

```
AccelerationStructure::AccelerationStructure ( )  [noexcept]
```

Definition at line 5 of file AccelerationStructure.cpp.

```
00005                                                        {
00006     m_Root = new BVHNode(new NonHittable());
00007 }
```

3.2.2.2 AccelerationStructure() [2/2]

```
AccelerationStructure::AccelerationStructure (
            std::span< HittableObjectPtr > objects )  [noexcept]
```

Definition at line 9 of file AccelerationStructure.cpp.

```
00009                                                                                      {
00010     Update(objects);
00011 }
```

3.2.2.3 ∼AccelerationStructure()

```
AccelerationStructure::∼AccelerationStructure ( )  [noexcept]
```

Definition at line 13 of file AccelerationStructure.cpp.

```
00013                                                       {
00014     if (m_Root != nullptr) {
00015         BVHNode::FreeMemory(m_Root);
00016     }
00017 }
```

3.2.3 Member Function Documentation

3.2.3.1 Hit()

```
void AccelerationStructure::Hit (
             const Ray & ray,
             float tMin,
             float tMax,
             HitPayload & payload ) const  [noexcept]
```

Definition at line 31 of file AccelerationStructure.cpp.

```
00031
      {
00032     m_Root->Hit(ray, tMin, tMax, payload);
00033 }
```

3.2.3.2 Update()

```
void AccelerationStructure::Update (
             std::span< HittableObjectPtr > objects )  [noexcept]
```

Definition at line 19 of file AccelerationStructure.cpp.

```
00019                                                                               {
00020     if (m_Root != nullptr) {
00021         BVHNode::FreeMemory(m_Root);
00022     }
00023
00024     if (objects.empty()) {
00025         m_Root = new BVHNode(new NonHittable());
00026     } else {
00027         m_Root = BVHNode::MakeHierarchySAH(objects, 0, static_cast<int>(objects.size()));
00028     }
00029 }
```

The documentation for this class was generated from the following files:

- AccelerationStructure.h
- AccelerationStructure.cpp

3.3 Application Class Reference

Path tracing application class which holds basic logic.

```
#include <Application.h>
```

Public Member Functions

- Application (int windowWidth, int windowHeight) noexcept
- int Run () noexcept

3.3.1 Detailed Description

Path tracing application class which holds basic logic.

Definition at line 15 of file Application.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Application()

```
Application::Application (
              int windowWidth,
              int windowHeight )  [noexcept]
```

Definition at line 21 of file Application.cpp.

```
00021                                                                       :
00022     m_InitialWindowWidth(windowWidth), m_InitialWindowHeight(windowHeight),
00023     m_LastViewportWidth(-1), m_LastViewportHeight(-1),
00024     m_Renderer(windowWidth, windowHeight),
00025     m_TotalRenderTime(0.f), m_LastRenderTime(0.f) {
00026 
00027     m_SaveImageFilePath = new char[c_AnyInputFilePathLength];
00028     m_SceneFilePath = new char[c_AnyInputFilePathLength];
00029     m_ModelFilePath = new char[c_AnyInputFilePathLength];
00030     m_MaterialDirectory = new char[c_AnyInputFilePathLength];
00031     memset(m_SaveImageFilePath, 0, sizeof(m_SaveImageFilePath));
00032     memset(m_SceneFilePath, 0, sizeof(m_SceneFilePath));
00033     memset(m_ModelFilePath, 0, sizeof(m_ModelFilePath));
00034     memset(m_MaterialDirectory, 0, sizeof(m_MaterialDirectory));
00035 
00036     m_AddMaterial.albedo = {0.f, 0.f, 0.f};
00037     m_AddMaterial.metallic = 0.f;
00038     m_AddMaterial.specular = 0.f;
00039     m_AddMaterial.roughness = 0.f;
00040     m_AddMaterial.emissionPower = 0.f;
00041     m_AddMaterial.index = -1;
00042 
00043     m_AddSphere = Shapes::Sphere(Math::Vector3f(0.f), 0.f, nullptr);
00044     m_AddTriangle = Shapes::Triangle(Math::Vector3f(0.f), Math::Vector3f(0.f), Math::Vector3f(0.f),
      nullptr);
00045     m_AddBox = Shapes::Box(Math::Vector3f(0.f), Math::Vector3f(0.f), nullptr);
00046 
00047     m_AddSphereMaterialIndex = -1;
00048     m_AddTriangleMaterialIndex = -1;
00049     m_AddBoxMaterialIndex = -1;
00050 
00051     m_RayMissColor = Math::Vector3f(0.f);
00052 
00053     m_Scene.camera = Camera(windowWidth, windowHeight);
00054 
00055     LoadSceneFromFile(c_DefaultScenePath);
00056 }
```

3.3.3 Member Function Documentation

3.3.3.1 Run()

```
int Application::Run ( ) [noexcept]
```

Definition at line 58 of file Application.cpp.

```
00058                               {
00059     if (!glfwInit()) {
00060         std::cerr « "Failed to initialize GLFW\n";
00061         return -1;
00062     }
00063 
00064 #if defined(IMGUI_IMPL_OPENGL_ES2)
00065     const char* glsl_version = "#version 100";
00066     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
00067     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
00068     glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_ES_API);
00069 #elif defined(__APPLE__)
00070     const char* glsl_version = "#version 150";
00071     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
00072     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
00073     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
00074     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
00075 #else
00076     const char* glsl_version = "#version 130";
00077     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
00078     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
00079 #endif
00080 
00081     m_Window = glfwCreateWindow(m_InitialWindowWidth, m_InitialWindowHeight, c_WindowTitle, nullptr,
      nullptr);
00082     if (m_Window == nullptr) {
00083         std::cerr « "Failed to create window\n";
00084         return -1;
00085     }
00086 
00087     GLFWimage *icon = new GLFWimage();
00088     icon->pixels = stbi_load("icon/icon.png", &icon->width, &icon->height, 0, 4);
00089     glfwSetWindowIcon(m_Window, 1, icon);
00090     stbi_image_free(icon->pixels);
00091     delete icon;
00092 
00093     glfwMakeContextCurrent(m_Window);
00094 
00095     ImGui::CreateContext();
00096     ImGuiIO &io = ImGui::GetIO();
00097     io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;
00098 
00099     UpdateThemeStyle();
00100 
00101     ImGui_ImplGlfw_InitForOpenGL(m_Window, true);
00102     ImGui_ImplOpenGL3_Init(glsl_version);
00103 
00104     MainLoop();
00105 
00106     ImGui_ImplOpenGL3_Shutdown();
00107     ImGui_ImplGlfw_Shutdown();
00108     ImGui::DestroyContext();
00109 
00110     glfwDestroyWindow(m_Window);
00111     glfwTerminate();
00112 
00113     return 0;
00114 }
```

The documentation for this class was generated from the following files:

- Application.h
- Application.cpp

3.4 BSDF Class Reference

Public Member Functions

- constexpr BSDF (const Material ∗material) noexcept
- Math::Vector3f Sample (const Ray &ray, const HitPayload &payload, Math::Vector3f &throughput) noexcept

3.4.1 Detailed Description

Definition at line 9 of file BSDF.h.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 BSDF()

```
constexpr BSDF::BSDF (
            const Material * material )  [inline], [constexpr], [noexcept]
```

Definition at line 11 of file BSDF.h.

```
00011                                                       :
00012        m_Material(material) {}
```

3.4.3 Member Function Documentation

3.4.3.1 Sample()

```
Math::Vector3f BSDF::Sample (
            const Ray & ray,
            const HitPayload & payload,
            Math::Vector3f & throughput )  [noexcept]
```

Definition at line 4 of file BSDF.cpp.

```
00004
      {
00005     float diffuseRatio = 0.5f * (1.f - m_Material->metallic);
00006     float specularRatio = 1.f - diffuseRatio;
00007
00008     Math::Vector3f V = -ray.direction;
00009
00010     Math::Vector3f reflectionDirection;
00011     if (Utilities::RandomFloatInZeroToOne() < diffuseRatio) {
00012         reflectionDirection = Utilities::RandomInHemisphere(payload.normal);
00013     } else {
00014         Math::Vector3f halfVec;
00015         {
00016             Math::Vector2f Xi{Utilities::RandomFloatInZeroToOne(),
      Utilities::RandomFloatInZeroToOne()};
00017             Math::Vector3f N = payload.normal;
00018
00019             float a = m_Material->roughness * m_Material->roughness;
00020
00021             float phi = Math::Constants::Tau<float> * Xi.x;
00022             float cosTheta = Math::Sqrt((1.f - Xi.y) / (1.f + (a * a - 1.f) * Xi.y));
00023             float sinTheta = Math::Sqrt(1.f - cosTheta * cosTheta);
00024
00025             Math::Vector3f H;
00026             H.x = Math::Cos(phi) * sinTheta;
00027             H.y = Math::Sin(phi) * sinTheta;
00028             H.z = cosTheta;
00029
00030             Math::Vector3f up = Math::Abs(N.z) < 0.999f ? Math::Vector3f(0.0, 0.0, 1.0) :
      Math::Vector3f(1.0, 0.0, 0.0);
00031             Math::Vector3f tangent = Math::Normalize(Math::Cross(up, N));
00032             Math::Vector3f bitangent = Math::Cross(N, tangent);
00033
00034             halfVec = tangent * H.x + bitangent * H.y + N * H.z;
00035             halfVec = Math::Normalize(halfVec);
00036         }
00037
00038         reflectionDirection = Math::Normalize(2.f * Math::Dot(V, halfVec) * halfVec - V);
00039     }
00040
00041     auto DistributionGGX = [](const Math::Vector3f &N, const Math::Vector3f &H, float roughness) {
00042         float a = roughness * roughness;
00043         float a2 = a * a;
```

```
        float NdotH = Math::Max(Math::Dot(N, H), 0.f);
        float NdotH2 = NdotH * NdotH;

        float nom = a2;
        float denom = (NdotH2 * (a2 - 1.f) + 1.f);
        denom = Math::Constants::Pi<float> * denom * denom;

        return nom / denom;
    };

    auto GeometrySchlickGGX = [](float NdotV, float roughness) {
        float r = (roughness + 1.f);
        float k = (r * r) / 8.f;

        float nom = NdotV;
        float denom = NdotV * (1.f - k) + k;

        return nom / denom;
    };

    auto GeometrySmith = [&](const Math::Vector3f &N, const Math::Vector3f &V, const Math::Vector3f &L, float roughness) {
        float NdotV = Math::Abs(Math::Dot(N, V));
        float NdotL = Math::Abs(Math::Dot(N, L));
        float ggx2 = GeometrySchlickGGX(NdotV, roughness);
        float ggx1 = GeometrySchlickGGX(NdotL, roughness);

        return ggx1 * ggx2;
    };

    auto FresnelSchlick = [](float cosTheta, const Math::Vector3f &F0) {
        return F0 + (1.f - F0) * Math::Pow(1.f - cosTheta, 5.f);
    };

    auto SpecularBRDF = [](float D, float G, const Math::Vector3f &F, const Math::Vector3f &V, const Math::Vector3f &L, const Math::Vector3f &N) {
        float NdotL = Math::Abs(Math::Dot(N, L));
        float NdotV = Math::Abs(Math::Dot(N, V));

        Math::Vector3f nominator = D * G * F;
        float denominator = 4.f * NdotV * NdotL + 0.001f;
        Math::Vector3f specularBrdf = nominator / denominator;

        return specularBrdf;
    };

    auto DiffuseBRDF = [](const Math::Vector3f &albedo) {
        return albedo * Math::Constants::InversePi<float>;
    };

    auto ImportanceSampleGGXPDF = [](float NDF, float NdotH, float VdotH) {
        return NDF * NdotH / (4.f * VdotH);
    };

    auto CosineSamplingPDF = [](float NdotL) {
        return NdotL * Math::Constants::InversePi<float>;
    };

    Math::Vector3f L = reflectionDirection;
    Math::Vector3f H = Math::Normalize(V + L);

    float NdotL = Math::Abs(Math::Dot(payload.normal, L));
    float NdotH = Math::Abs(Math::Dot(payload.normal, H));
    float VdotH = Math::Abs(Math::Dot(V, H));

    float NdotV = Math::Abs(Math::Dot(payload.normal, V));

    Math::Vector3f F0 = Math::Vector3f(0.08f, 0.08f, 0.08f);
    F0 = Math::Lerp(F0 * m_Material->specular, m_Material->albedo, m_Material->metallic);

    float NDF = DistributionGGX(payload.normal, H, m_Material->roughness);
    float G = GeometrySmith(payload.normal, V, L, m_Material->roughness);
    Math::Vector3f F = FresnelSchlick(Math::Max(Math::Dot(H, V), 0.f), F0);

    Math::Vector3f kS = F;
    Math::Vector3f kD = 1.f - kS;
    kD *= 1.0 - m_Material->metallic;

    Math::Vector3f specularBrdf = SpecularBRDF(NDF, G, F, V, L, payload.normal);

    float specularPdf = ImportanceSampleGGXPDF(NDF, NdotH, VdotH);

    Math::Vector3f diffuseBrdf = DiffuseBRDF(m_Material->albedo);
    float diffusePdf = CosineSamplingPDF(NdotL);

    Math::Vector3f totalBrdf = (diffuseBrdf * kD + specularBrdf) * NdotL;
    float totalPdf = diffuseRatio * diffusePdf + specularRatio * specularPdf;

    if (totalPdf > Math::Constants::Epsilon<float>) {
        throughput *= totalBrdf / totalPdf;
    }

    return reflectionDirection;
}
```

The documentation for this class was generated from the following files:

- BSDF.h
- BSDF.cpp

3.5 BVHNode Struct Reference

Public Member Functions

- constexpr BVHNode (BVHNode *left, BVHNode *right) noexcept
- constexpr BVHNode (const HittableObject *object) noexcept
- constexpr bool IsTerminating () const noexcept
- bool Hit (const Ray &ray, float tMin, float tMax, HitPayload &payload) noexcept

Static Public Member Functions

- static BVHNode * MakeHierarchySAH (std::span< HittableObjectPtr > objects, int low, int high) noexcept
- static BVHNode * MakeHierarchyNaive (std::span< HittableObjectPtr > objects, int low, int high) noexcept
- static constexpr void FreeMemory (BVHNode *node) noexcept

Public Attributes

- AABB aabb = AABB::Empty()
- BVHNode * left = nullptr
- BVHNode * right = nullptr
- const HittableObject * object = nullptr

Static Public Attributes

- static const std::function< bool(HittableObjectPtr, HittableObjectPtr)> c_ComparatorsSAH [3]
- static const std::function< bool(HittableObjectPtr, HittableObjectPtr)> c_ComparatorsNaive [3]

3.5.1 Detailed Description

Definition at line 13 of file BVHNode.h.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 BVHNode() [1/2]

```
constexpr BVHNode::BVHNode (
            BVHNode * left,
            BVHNode * right ) [inline], [constexpr], [noexcept]
```

Definition at line 20 of file BVHNode.h.

```
00020                                                                        :
00021         left(left), right(right), aabb(left->aabb, right->aabb) {}
```

3.5.2.2 BVHNode() [2/2]

```
constexpr BVHNode::BVHNode (
            const HittableObject * object ) [inline], [constexpr], [noexcept]
```

Definition at line 23 of file BVHNode.h.

```
00023                                                                       :
00024         object(object), aabb(object->GetBoundingBox()) {}
```

3.5.3 Member Function Documentation

3.5.3.1 FreeMemory()

```
static constexpr void BVHNode::FreeMemory (
            BVHNode * node ) [inline], [static], [constexpr], [noexcept]
```

Definition at line 154 of file BVHNode.h.

```
00154                                                                       {
00155         if (node == nullptr) {
00156             return;
00157         }
00158 
00159         FreeMemory(node->left);
00160         FreeMemory(node->right);
00161 
00162         delete node;
00163     }
```

3.5.3.2 Hit()

```
bool BVHNode::Hit (
            const Ray & ray,
            float tMin,
            float tMax,
            HitPayload & payload ) [inline], [noexcept]
```

Definition at line 30 of file BVHNode.h.

```
00030                                                                                                {
00031         if (!aabb.IntersectsRay(ray, tMin, tMax)) {
00032             return false;
00033         }
00034 
00035         if (IsTerminating()) {
00036             return object->Hit(ray, tMin, tMax, payload);
00037         }
00038 
00039         bool anyHit = left->Hit(ray, tMin, tMax, payload);
00040         anyHit |= right->Hit(ray, tMin, Math::Min(tMax, payload.t), payload);
00041 
00042         return anyHit;
00043     }
```

3.5.3.3 IsTerminating()

```
constexpr bool BVHNode::IsTerminating ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 26 of file BVHNode.h.

```
00026                                                     {
00027         return object != nullptr;
00028     }
```

3.5.3.4 MakeHierarchyNaive()

```
static BVHNode * BVHNode::MakeHierarchyNaive (
              std::span< HittableObjectPtr > objects,
              int low,
              int high )  [inline], [static], [noexcept]
```

Definition at line 108 of file BVHNode.h.

```
00108
      {
00109         AABB aabb = AABB::Empty();
00110         for (int i = low; i < high; ++i) {
00111             aabb = AABB(aabb, objects[i]->GetBoundingBox());
00112         }
00113
00114         int longestAxisIndex = 0;
00115         float longestAxisLength = aabb.max.x - aabb.min.x;
00116
00117         if (aabb.max.y - aabb.min.y > longestAxisLength) {
00118             longestAxisIndex = 1;
00119             longestAxisLength = aabb.max.y - aabb.min.y;
00120         }
00121
00122         if (aabb.max.z - aabb.min.z > longestAxisLength) {
00123             longestAxisIndex = 2;
00124             longestAxisLength = aabb.max.z - aabb.min.z;
00125         }
00126
00127         auto comparator = c_ComparatorsNaive[longestAxisIndex];
00128
00129         if (low + 1 == high) {
00130             return new BVHNode(objects[low]);
00131         }
00132
00133         std::sort(objects.begin() + low, objects.begin() + high, comparator);
00134
00135         int mid = (low + high) / 2;
00136         BVHNode *left = MakeHierarchyNaive(objects, low, mid);
00137         BVHNode *right = MakeHierarchyNaive(objects, mid, high);
00138
00139         return new BVHNode(left, right);
00140     }
```

3.5.3.5 MakeHierarchySAH()

```
static BVHNode * BVHNode::MakeHierarchySAH (
              std::span< HittableObjectPtr > objects,
              int low,
              int high )  [inline], [static], [noexcept]
```

Definition at line 45 of file BVHNode.h.

```
00045
      {
00046         if (low + 1 == high) {
00047             return new BVHNode(objects[low]);
00048         }
00049
00050         int n = high - low;
00051         std::vector<AABB> pref(n + 1);
00052         std::vector<AABB> suff(n + 1);
00053
```

```
        float minValue = Math::Constants::Infinity<float>;
        int mid = -1;
        int dimension = -1;

        for (int d = 0; d < 3; ++d) {
            std::sort(objects.begin() + low, objects.begin() + high, c_ComparatorsSAH[d]);

            pref[0] = AABB::Empty();
            for (int i = 0; i < n; ++i) {
                pref[i + 1] = AABB(pref[i], objects[i + low]->GetBoundingBox());
            }

            suff[n] = AABB::Empty();
            for (int i = n - 1; i >= 0; --i) {
                suff[i] = AABB(objects[i + low]->GetBoundingBox(), suff[i + 1]);
            }

            float minValueAlongAxis = Math::Constants::Infinity<float>;
            int index = -1;
            for (int i = 0; i < n; ++i) {
                float value = pref[i + 1].GetSurfaceArea() * (float)(i + 1) + suff[i +
1].GetSurfaceArea() * (float)(n - i - 1);
                if (value < minValueAlongAxis) {
                    minValueAlongAxis = value;
                    index = i + low;
                }
            }

            if (minValueAlongAxis < minValue) {
                minValue = minValueAlongAxis;
                mid = index + 1;
                dimension = d;
            }
        }

        std::sort(objects.begin() + low, objects.begin() + high, c_ComparatorsSAH[dimension]);

        BVHNode *left = MakeHierarchySAH(objects, low, mid);
        BVHNode *right = MakeHierarchySAH(objects, mid, high);

        return new BVHNode(left, right);
    }
```

3.5.4 Member Data Documentation

3.5.4.1 aabb

```
AABB BVHNode::aabb = AABB::Empty()
```

Definition at line 14 of file BVHNode.h.

3.5.4.2 c_ComparatorsNaive

```
const std::function<bool(HittableObjectPtr, HittableObjectPtr)> BVHNode::c_ComparatorsNaive[3]
[inline], [static]
```

Initial value:

```
= {
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.x < b->GetBoundingBox().min.x;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.y < b->GetBoundingBox().min.y;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.z < b->GetBoundingBox().min.z;
        }
    }
```

Definition at line 142 of file BVHNode.h.

```
00142
      {
```

```
00143         [](HittableObjectPtr a, HittableObjectPtr b){
00144             return a->GetBoundingBox().min.x < b->GetBoundingBox().min.x;
00145         },
00146         [](HittableObjectPtr a, HittableObjectPtr b){
00147             return a->GetBoundingBox().min.y < b->GetBoundingBox().min.y;
00148         },
00149         [](HittableObjectPtr a, HittableObjectPtr b){
00150             return a->GetBoundingBox().min.z < b->GetBoundingBox().min.z;
00151         }
00152     };
```

3.5.4.3 c_ComparatorsSAH

```
const std::function<bool(HittableObjectPtr, HittableObjectPtr)> BVHNode::c_ComparatorsSAH[3]
[inline], [static]
```

Initial value:

```
= {
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().x < b->GetCentroid().x;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().y < b->GetCentroid().y;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().z < b->GetCentroid().z;
        }
    }
```

Definition at line 96 of file BVHNode.h.

```
00096
      {
00097         [](HittableObjectPtr a, HittableObjectPtr b){
00098             return a->GetCentroid().x < b->GetCentroid().x;
00099         },
00100         [](HittableObjectPtr a, HittableObjectPtr b){
00101             return a->GetCentroid().y < b->GetCentroid().y;
00102         },
00103         [](HittableObjectPtr a, HittableObjectPtr b){
00104             return a->GetCentroid().z < b->GetCentroid().z;
00105         }
00106     };
```

3.5.4.4 left

```
BVHNode* BVHNode::left = nullptr
```

Definition at line 15 of file BVHNode.h.

3.5.4.5 object

```
const HittableObject* BVHNode::object = nullptr
```

Definition at line 16 of file BVHNode.h.

3.5.4.6 right

```
BVHNode * BVHNode::right = nullptr
```

Definition at line 15 of file BVHNode.h.

The documentation for this struct was generated from the following file:

- BVHNode.h

3.6 Camera Class Reference

Public Member Functions

- Camera (int viewportWidth, int viewportHeight, const Math::Vector3f &position={0.f, 0.f, 0.f}, const Math::Vector3f &target={0.f, 0.f, -1.f}, float verticalFovInDegrees=20.f, const Math::Vector3f &up={0.f, 1.f, 0.f}) noexcept
- void OnViewportResize (int viewportWidth, int viewportHeight) noexcept
- void ComputeRayDirections () noexcept
- constexpr std::span< const Math::Vector3f > GetRayDirections () const noexcept
- constexpr Math::Vector3f GetPosition () const noexcept
- constexpr Math::Vector3f & Position () noexcept
- constexpr Math::Vector3f GetTarget () const noexcept
- constexpr Math::Vector3f & Target () noexcept
- constexpr Math::Vector3f GetUp () const noexcept
- constexpr Math::Vector3f & Up () noexcept
- constexpr float GetVerticalFovInDegrees () const noexcept
- constexpr float & VerticalFovInDegrees () noexcept

3.6.1 Detailed Description

Definition at line 9 of file Camera.h.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Camera()

```
Camera::Camera (
              int viewportWidth,
              int viewportHeight,
              const Math::Vector3f & position = {0.f, 0.f, 0.f},
              const Math::Vector3f & target = {0.f, 0.f, -1.f},
              float verticalFovInDegrees = 20.f,
              const Math::Vector3f & up = {0.f, 1.f, 0.f} )  [noexcept]
```

Definition at line 4 of file Camera.cpp.

```
00004
      :
00005     m_ViewportWidth(viewportWidth), m_ViewportHeight(viewportHeight), m_Position(position),
      m_Target(target), m_VerticalFovInDegrees(verticalFovInDegrees), m_Up(up) {
00006     m_RayDirections.resize(m_ViewportWidth * m_ViewportHeight);
00007     ComputeRayDirections();
00008 }
```

3.6.3 Member Function Documentation

3.6.3.1 ComputeRayDirections()

```
void Camera::ComputeRayDirections ( )  [noexcept]
```

Definition at line 21 of file Camera.cpp.

```
00021                                      {
00022     float verticalFovInRadians = Math::ToRadians(m_VerticalFovInDegrees);
00023 
00024     Math::Vector3f forward = m_Target - m_Position;
00025     float focalLength = Math::Length(forward);
00026     float viewportWorldHeight = 2.f * Math::Tan(verticalFovInRadians * 0.5f) * focalLength;
00027     float viewportWorldWidth = (viewportWorldHeight * m_ViewportWidth) / m_ViewportHeight;
00028 
00029     Math::Vector3f w = Math::Normalize(forward);
00030     Math::Vector3f u = Math::Normalize(Math::Cross(w, m_Up));
00031     Math::Vector3f v = Math::Normalize(Math::Cross(w, u));
00032 
00033     Math::Vector3f horizontal = u * viewportWorldWidth;
00034     Math::Vector3f vertical = v * viewportWorldHeight;
00035     Math::Vector3f leftUpper = forward - horizontal * 0.5f - vertical * 0.5f;
00036 
00037     for (int i = 0; i < m_ViewportHeight; ++i) {
00038         for (int j = 0; j < m_ViewportWidth; ++j) {
00039             float uScale = (float)(j + Utilities::RandomFloatInNegativeHalfToHalf()) /
      (m_ViewportWidth - 1);
00040             float vScale = (float)(i + Utilities::RandomFloatInNegativeHalfToHalf()) /
      (m_ViewportHeight - 1);
00041             m_RayDirections[m_ViewportWidth * i + j] = Math::Normalize(leftUpper + horizontal * uScale
      + vertical * vScale);
00042         }
00043     }
00044 }
```

3.6.3.2 GetPosition()

```
constexpr Math::Vector3f Camera::GetPosition ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 25 of file Camera.h.

```
00025                                                     {
00026         return m_Position;
00027     }
```

3.6.3.3 GetRayDirections()

```
constexpr std::span< const Math::Vector3f > Camera::GetRayDirections ( ) const  [inline],
[constexpr], [noexcept]
```

Definition at line 21 of file Camera.h.

```
00021                                                                            {
00022         return m_RayDirections;
00023     }
```

3.6.3.4 GetTarget()

```
constexpr Math::Vector3f Camera::GetTarget ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 33 of file Camera.h.

```
00033                                                   {
00034         return m_Target;
00035     }
```

3.6.3.5 GetUp()

```
constexpr Math::Vector3f Camera::GetUp ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 41 of file Camera.h.

```
00041                                                {
00042         return m_Up;
00043     }
```

3.6.3.6 GetVerticalFovInDegrees()

```
constexpr float Camera::GetVerticalFovInDegrees ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 49 of file Camera.h.

```
00049                                                           {
00050         return m_VerticalFovInDegrees;
00051     }
```

3.6.3.7 OnViewportResize()

```
void Camera::OnViewportResize (
             int viewportWidth,
             int viewportHeight )  [noexcept]
```

Definition at line 10 of file Camera.cpp.

```
00010                                                                          {
00011     if (m_ViewportWidth == viewportWidth && m_ViewportHeight == viewportHeight) {
00012         return;
00013     }
00014 
00015     m_ViewportWidth = viewportWidth;
00016     m_ViewportHeight = viewportHeight;
00017 
00018     m_RayDirections.resize(m_ViewportWidth * m_ViewportHeight);
00019 }
```

3.6.3.8 Position()

```
constexpr Math::Vector3f & Camera::Position ( )  [inline], [constexpr], [noexcept]
```

Definition at line 29 of file Camera.h.

```
00029                                              {
00030         return m_Position;
00031     }
```

3.6.3.9 Target()

```
constexpr Math::Vector3f & Camera::Target ( )  [inline], [constexpr], [noexcept]
```

Definition at line 37 of file Camera.h.

```
00037                                            {
00038         return m_Target;
00039     }
```

3.6.3.10 Up()

```
constexpr Math::Vector3f & Camera::Up ( ) [inline], [constexpr], [noexcept]
```

Definition at line 45 of file Camera.h.

```
00045                                        {
00046         return m_Up;
00047     }
```

3.6.3.11 VerticalFovInDegrees()

```
constexpr float & Camera::VerticalFovInDegrees ( ) [inline], [constexpr], [noexcept]
```

Definition at line 53 of file Camera.h.

```
00053                                                      {
00054         return m_VerticalFovInDegrees;
00055     }
```

The documentation for this class was generated from the following files:

- Camera.h
- Camera.cpp

3.7 HitPayload Struct Reference

Public Attributes

- float t
- Math::Vector3f normal
- const Material ∗ material

3.7.1 Detailed Description

Definition at line 7 of file HitPayload.h.

3.7.2 Member Data Documentation

3.7.2.1 material

```
const Material* HitPayload::material
```

Definition at line 10 of file HitPayload.h.

3.7.2.2 normal

```
Math::Vector3f HitPayload::normal
```

Definition at line 9 of file HitPayload.h.

3.7.2.3 t

```
float HitPayload::t
```

Definition at line 8 of file HitPayload.h.

The documentation for this struct was generated from the following file:

- HitPayload.h

3.8 Image Class Reference

Public Member Functions

- Image (int width, int height, const uint32_t ∗data) noexcept
- void UpdateData (const uint32_t ∗data) noexcept
- constexpr unsigned int GetDescriptor () const noexcept

3.8.1 Detailed Description

Definition at line 7 of file Image.h.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 Image()

```
Image::Image (
              int width,
              int height,
              const uint32_t * data )  [noexcept]
```

Definition at line 4 of file Image.cpp.

```
00004                                                                    :
00005     m_Width(width), m_Height(height) {
00006     glGenTextures(1, &m_Descriptor);
00007     glBindTexture(GL_TEXTURE_2D, m_Descriptor);
00008 
00009     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
00010     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
00011 
00012     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_Width, m_Height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
      (void*)data);
00013 
00014     glBindTexture(GL_TEXTURE_2D, 0);
00015 }
```

3.8.2.2 ∼Image()

```
Image::∼Image ( )  [noexcept]
```

Definition at line 17 of file Image.cpp.

```
00017                        {
00018     glDeleteTextures(1, &m_Descriptor);
00019 }
```

3.8.3 Member Function Documentation

3.8.3.1 GetDescriptor()

```
constexpr unsigned int Image::GetDescriptor ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 17 of file Image.h.

```
00017                                                            {
00018         return m_Descriptor;
00019     }
```

3.8.3.2 UpdateData()

```
void Image::UpdateData (
              const uint32_t * data )  [noexcept]
```

Definition at line 21 of file Image.cpp.

```
00021                                                       {
00022     glBindTexture(GL_TEXTURE_2D, m_Descriptor);
00023 
00024     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
00025     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
00026 
00027     glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, m_Width, m_Height, GL_RGBA, GL_UNSIGNED_BYTE,
      (void*)data);
00028 
00029     glBindTexture(GL_TEXTURE_2D, 0);
00030 }
```

The documentation for this class was generated from the following files:

- Image.h
- Image.cpp

3.9 Light Class Reference

Public Member Functions

- constexpr Light (const HittableObjectPtr object, const Math::Vector3f &emission) noexcept
- const HittableObject * GetObject () const noexcept
- Math::Vector3f Sample (const Ray &lightRay, const HitPayload &objectHitPayload, const HitPayload &light←HitPayload, float distance, float distanceSquared) const noexcept

3.9.1 Detailed Description

Definition at line 9 of file Light.h.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 Light()

```
constexpr Light::Light (
              const HittableObjectPtr object,
              const Math::Vector3f & emission )  [inline], [constexpr], [noexcept]
```

Definition at line 11 of file Light.h.

```
00011                                                                                          :
00012         m_Object(object), m_Emission(emission) {}
```

3.9.3 Member Function Documentation

3.9.3.1 GetObject()

```
const HittableObject * Light::GetObject ( ) const  [inline], [noexcept]
```

Definition at line 14 of file Light.h.

```
00014                                                     {
00015         return m_Object;
00016     }
```

3.9.3.2 Sample()

```
Math::Vector3f Light::Sample (
             const Ray & lightRay,
             const HitPayload & objectHitPayload,
             const HitPayload & lightHitPayload,
             float distance,
             float distanceSquared ) const  [inline], [noexcept]
```

Definition at line 18 of file Light.h.

```
00018
      {
00019         constexpr float distanceEpsilon = 0.01f;
00020         if (Math::Abs(lightHitPayload.t - distance) > distanceEpsilon) {
00021             return Math::Vector3f(0.f);
00022         }
00023
00024         float pdf = distanceSquared / (Math::Dot(lightHitPayload.normal, -lightRay.direction) *
      m_Object->GetArea());
00025
00026         constexpr float pdfEpsilon = 0.01f;
00027         if (pdf <= pdfEpsilon) {
00028             return Math::Vector3f(0.f);
00029         }
00030
00031         Math::Vector3f brdf = objectHitPayload.material->albedo * Math::Constants::InversePi<float> *
      Math::Dot(objectHitPayload.normal, lightRay.direction) * m_Emission;
00032
00033         return brdf / pdf;
00034     }
```

The documentation for this class was generated from the following file:

- Light.h

3.10 Material Struct Reference

Public Member Functions

- constexpr Math::Vector3f GetEmission () const noexcept

Public Attributes

- Math::Vector3f albedo
- float metallic
- float specular
- float roughness
- float emissionPower
- int index

3.10.1 Detailed Description

Definition at line 6 of file Material.h.

3.10.2 Member Function Documentation

3.10.2.1 GetEmission()

```
constexpr Math::Vector3f Material::GetEmission ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 15 of file Material.h.

```
00015                                                    {
00016         return albedo * emissionPower;
00017     }
```

3.10.3 Member Data Documentation

3.10.3.1 albedo

```
Math::Vector3f Material::albedo
```

Definition at line 7 of file Material.h.

3.10.3.2 emissionPower

```
float Material::emissionPower
```

Definition at line 11 of file Material.h.

3.10.3.3 index

```
int Material::index
```

Definition at line 13 of file Material.h.

3.10.3.4 metallic

```
float Material::metallic
```

Definition at line 8 of file Material.h.

3.10.3.5 roughness

```
float Material::roughness
```

Definition at line 10 of file Material.h.

3.10.3.6 specular

```
float Material::specular
```

Definition at line 9 of file Material.h.

The documentation for this struct was generated from the following file:

- Material.h

3.11 Ray Struct Reference

Public Attributes

- Math::Vector3f origin
- Math::Vector3f direction

3.11.1 Detailed Description

Definition at line 6 of file Ray.h.

3.11.2 Member Data Documentation

3.11.2.1 direction

```
Math::Vector3f Ray::direction
```

Definition at line 7 of file Ray.h.

3.11.2.2 origin

```
Math::Vector3f Ray::origin
```

Definition at line 7 of file Ray.h.

The documentation for this struct was generated from the following file:

- Ray.h

3.12 Renderer Class Reference

Public Types

- using typ_t = HittableObject*

Public Member Functions

- Renderer (int width, int height) noexcept
- void Render (const Camera &camera, std::span< const HittableObjectPtr > objects, std::span< const Light > lightSources, std::span< const Material > materials) noexcept
- void Render (const Camera &camera, const AccelerationStructure &accelerationStructure, std::span< const Light > lightSources, std::span< const Material > materials) noexcept
- constexpr bool & Accumulate () noexcept
- constexpr bool & Accelerate () noexcept
- constexpr int GetFrameIndex () const noexcept
- constexpr Image * GetImage () const noexcept
- void SaveImage (const char *filename) const noexcept
- void OnResize (int width, int height) noexcept
- constexpr int GetAvailableThreadCount () const noexcept
- constexpr int GetUsedThreadCount () const noexcept
- constexpr void SetUsedThreadCount (int usedThreads) noexcept
- constexpr int & UsedThreadCount () noexcept
- void OnRayMiss (std::function< Math::Vector3f(const Ray &)> onRayMiss) noexcept
- constexpr int & RayDepth () noexcept
- constexpr float & Gamma () noexcept

3.12.1 Detailed Description

Definition at line 17 of file Renderer.h.

3.12.2 Member Typedef Documentation

3.12.2.1 typ_t

```
using Renderer::typ_t = HittableObject*
```

Definition at line 25 of file Renderer.h.

3.12.3 Constructor & Destructor Documentation

3.12.3.1 Renderer()

```
Renderer::Renderer (
              int width,
              int height )   [noexcept]
```

Definition at line 12 of file Renderer.cpp.

```
00012                                                 :
00013     m_Width(width), m_Height(height) {
00014     m_ImageData = new uint32_t[m_Width * m_Height];
00015     m_Image = new Image(m_Width, m_Height, m_ImageData);
00016     m_AccumulationData = new Math::Vector4f[m_Width * m_Height];
00017 
00018     m_AvailableThreads = std::thread::hardware_concurrency();
00019     m_UsedThreads = 1;
00020     m_LinesPerThread = (m_Height + m_UsedThreads - 1) / m_UsedThreads;
00021 }
```

3.12.3.2 ∼Renderer()

```
Renderer::∼Renderer ( )  [noexcept]
```

Definition at line 23 of file Renderer.cpp.

```
00023                                    {
00024     if (m_Image != nullptr) {
00025         delete m_Image;
00026     }
00027     if (m_ImageData != nullptr) {
00028         delete[] m_ImageData;
00029     }
00030     if (m_AccumulationData != nullptr) {
00031         delete[] m_AccumulationData;
00032     }
00033 }
```

3.12.4 Member Function Documentation

3.12.4.1 Accelerate()

```
constexpr bool & Renderer::Accelerate ( )  [inline], [constexpr], [noexcept]
```

Definition at line 35 of file Renderer.h.

```
00035                                                 {
00036         return m_Accelerate;
00037     }
```

3.12.4.2 Accumulate()

```
constexpr bool & Renderer::Accumulate ( )  [inline], [constexpr], [noexcept]
```

Definition at line 31 of file Renderer.h.

```
00031                                                 {
00032         return m_Accumulate;
00033     }
```

3.12.4.3 Gamma()

```
constexpr float & Renderer::Gamma ( )  [inline], [constexpr], [noexcept]
```

Definition at line 76 of file Renderer.h.

```
00076                                             {
00077         return m_Gamma;
00078     }
```

3.12.4.4 GetAvailableThreadCount()

```
constexpr int Renderer::GetAvailableThreadCount ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 51 of file Renderer.h.

```
00051                                                                   {
00052         return m_AvailableThreads;
00053     }
```

3.12.4.5 GetFrameIndex()

```
constexpr int Renderer::GetFrameIndex ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 39 of file Renderer.h.

```
00039                                                   {
00040         return m_FrameIndex;
00041     }
```

3.12.4.6 GetImage()

```
constexpr Image * Renderer::GetImage ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 43 of file Renderer.h.

```
00043                                                 {
00044         return m_Image;
00045     }
```

3.12.4.7 GetUsedThreadCount()

```
constexpr int Renderer::GetUsedThreadCount ( ) const  [inline], [constexpr], [noexcept]
```

Definition at line 55 of file Renderer.h.

```
00055                                                        {
00056         return m_UsedThreads;
00057     }
```

3.12.4.8 OnRayMiss()

```
void Renderer::OnRayMiss (
              std::function< Math::Vector3f(const Ray &)> onRayMiss )  [inline], [noexcept]
```

Definition at line 68 of file Renderer.h.

```
00068                                                                                      {
00069         m_OnRayMiss = onRayMiss;
00070     }
```

3.12.4.9 OnResize()

```
void Renderer::OnResize (
              int width,
              int height )  [noexcept]
```

Definition at line 39 of file Renderer.cpp.

```
00039                                                        {
00040     if (m_Width == width && m_Height == height) {
00041         return;
00042     }
00043 
00044     m_Width = width;
00045     m_Height = height;
00046 
00047     if (m_ImageData != nullptr) {
00048         delete m_ImageData;
00049         m_ImageData = new uint32_t[m_Width * m_Height];
00050     }
00051     if (m_Image != nullptr) {
00052         delete m_Image;
00053         m_Image = new Image(m_Width, m_Height, m_ImageData);
00054     }
00055     if (m_AccumulationData != nullptr) {
00056         delete m_AccumulationData;
00057         m_AccumulationData = new Math::Vector4f[m_Width * m_Height];
00058     }
00059 }
```

3.12.4.10 RayDepth()

```
constexpr int & Renderer::RayDepth ( )  [inline], [constexpr], [noexcept]
```

Definition at line 72 of file Renderer.h.

```
00072                                       {
00073         return m_RayDepth;
00074     }
```

3.12.4.11 Render() [1/2]

```
void Renderer::Render (
              const Camera & camera,
              const AccelerationStructure & accelerationStructure,
              std::span< const Light > lightSources,
              std::span< const Material > materials )  [noexcept]
```

Definition at line 112 of file Renderer.cpp.

```
00112
      {
00113     m_Camera = &camera;
00114     m_AccelerationStructure = &accelerationStructure;
00115     m_LightSources = lightSources;
00116     m_Materials = materials;
00117
00118     if (!m_Accumulate) {
00119         m_FrameIndex = 1;
00120     }
00121
00122     if (m_FrameIndex == 1) {
00123         memset(m_AccumulationData, 0, m_Width * m_Height * sizeof(Math::Vector4f));
00124     }
00125
00126     float inverseFrameIndex = 1.f / m_FrameIndex;
00127     float inverseGamma = 1.f / m_Gamma;
00128
00129     std::vector<std::thread> handles;
00130     handles.reserve(m_UsedThreads);
00131
00132     for (int i = 0; i < m_Height; i += m_LinesPerThread) {
00133         handles.emplace_back([this, i, inverseFrameIndex, inverseGamma]() {
00134             int nextBlock = i + m_LinesPerThread;
00135             int limit = Math::Min(nextBlock, m_Height);
00136             for (int t = i; t < limit; ++t) {
00137                 for (int j = 0; j < m_Width; ++j) {
00138                     m_AccumulationData[m_Width * t + j] += AcceleratedPixelProgram(t, j);
00139
00140                     Math::Vector4f color = m_AccumulationData[m_Width * t + j];
00141
00142                     color *= inverseFrameIndex;
00143                     color = Utilities::CorrectGamma(color, inverseGamma);
00144                     color = Math::Clamp(color, 0.f, 1.f);
00145
00146                     m_ImageData[m_Width * t + j] = Utilities::ConvertColorToRGBA(color);
00147                 }
00148             }
00149         });
00150     }
00151
00152     for (auto &handle : handles) {
00153         handle.join();
00154     }
00155
00156     m_Image->UpdateData(m_ImageData);
00157
00158     if (m_Accumulate) {
00159         ++m_FrameIndex;
00160     }
00161 }
```

3.12.4.12 Render() [2/2]

```
void Renderer::Render (
               const Camera & camera,
               std::span< const HittableObjectPtr > objects,
               std::span< const Light > lightSources,
               std::span< const Material > materials )   [noexcept]
```

Definition at line 61 of file Renderer.cpp.

```
00061
      {
00062     m_Camera = &camera;
00063     m_Objects = objects;
00064     m_LightSources = lightSources;
00065     m_Materials = materials;
00066
00067     if (!m_Accumulate) {
00068         m_FrameIndex = 1;
00069     }
00070
00071     if (m_FrameIndex == 1) {
00072         memset(m_AccumulationData, 0, m_Width * m_Height * sizeof(Math::Vector4f));
00073     }
00074
00075     float inverseFrameIndex = 1.f / m_FrameIndex;
00076     float inverseGamma = 1.f / m_Gamma;
00077
00078     std::vector<std::thread> handles;
00079     handles.reserve(m_UsedThreads);
00080
00081     for (int i = 0; i < m_Height; i += m_LinesPerThread) {
00082         handles.emplace_back([this, i, inverseFrameIndex, inverseGamma]() {
00083             int nextBlock = i + m_LinesPerThread;
00084             int limit = Math::Min(nextBlock, m_Height);
00085             for (int t = i; t < limit; ++t) {
00086                 for (int j = 0; j < m_Width; ++j) {
00087                     m_AccumulationData[m_Width * t + j] += PixelProgram(t, j);
00088
00089                     Math::Vector4f color = m_AccumulationData[m_Width * t + j];
00090
00091                     color *= inverseFrameIndex;
00092                     color = Utilities::CorrectGamma(color, inverseGamma);
00093                     color = Math::Clamp(color, 0.f, 1.f);
00094
00095                     m_ImageData[m_Width * t + j] = Utilities::ConvertColorToRGBA(color);
00096                 }
00097             }
00098         });
00099     }
00100
00101     for (auto &handle : handles) {
00102         handle.join();
00103     }
00104
00105     m_Image->UpdateData(m_ImageData);
00106
00107     if (m_Accumulate) {
00108         ++m_FrameIndex;
00109     }
00110 }
```

3.12.4.13 SaveImage()

```
void Renderer::SaveImage (
               const char * filename ) const   [noexcept]
```

Definition at line 35 of file Renderer.cpp.

```
00035                                                             {
00036     stbi_write_png(filename, m_Width, m_Height, 4, m_ImageData, m_Width * 4);
00037 }
```

3.12.4.14 SetUsedThreadCount()

```
constexpr void Renderer::SetUsedThreadCount (
              int usedThreads )  [inline], [constexpr], [noexcept]
```

Definition at line 59 of file Renderer.h.

```
00059                                                                           {
00060         m_UsedThreads = Math::Clamp(usedThreads, 1, m_AvailableThreads);
00061         m_LinesPerThread = (m_Height + m_UsedThreads - 1) / m_UsedThreads;
00062     }
```

3.12.4.15 UsedThreadCount()

```
constexpr int & Renderer::UsedThreadCount ( )  [inline], [constexpr], [noexcept]
```

Definition at line 64 of file Renderer.h.

```
00064                                                         {
00065         return m_UsedThreads;
00066     }
```

The documentation for this class was generated from the following files:

- Renderer.h
- Renderer.cpp

3.13 Scene Struct Reference

Public Member Functions

- std::optional< std::string > Serialize (std::ostream &os) const noexcept
- std::optional< std::string > Deserialize (std::istream &is) noexcept

Public Attributes

- std::vector< Shapes::Sphere > spheres
- std::vector< Shapes::Triangle > triangles
- std::vector< Shapes::Box > boxes
- std::vector< Model * > models
- std::vector< Material > materials
- Camera camera

3.13.1 Detailed Description

Definition at line 17 of file Scene.h.

3.13.2 Member Function Documentation

3.13.2.1 Deserialize()

```
std::optional< std::string > Scene::Deserialize (
            std::istream & is )  [inline], [noexcept]
```

Definition at line 37 of file Scene.h.

```
00037                                                           {
00038         is.exceptions(std::ios::eofbit | std::ios::badbit | std::ios::failbit);
00039 
00040         try {
00041             TryDeserialize(is);
00042         } catch (std::exception &e) {
00043             return e.what();
00044         }
00045 
00046         return {};
00047     }
```

3.13.2.2 Serialize()

```
std::optional< std::string > Scene::Serialize (
            std::ostream & os ) const  [inline], [noexcept]
```

Definition at line 25 of file Scene.h.

```
00025                                                                 {
00026         os.exceptions(std::ios::badbit | std::ios::failbit);
00027 
00028         try {
00029             TrySerialize(os);
00030         } catch (std::exception &e) {
00031             return e.what();
00032         }
00033 
00034         return {};
00035     }
```

3.13.3 Member Data Documentation

3.13.3.1 boxes

```
std::vector<Shapes::Box> Scene::boxes
```

Definition at line 20 of file Scene.h.

3.13.3.2 camera

```
Camera Scene::camera
```

Definition at line 23 of file Scene.h.

3.13.3.3 materials

```
std::vector<Material> Scene::materials
```

Definition at line 22 of file Scene.h.

3.13.3.4 models

`std::vector<Model*> Scene::models`

Definition at line 21 of file Scene.h.

3.13.3.5 spheres

`std::vector<Shapes::Sphere> Scene::spheres`

Definition at line 18 of file Scene.h.

3.13.3.6 triangles

`std::vector<Shapes::Triangle> Scene::triangles`

Definition at line 19 of file Scene.h.

The documentation for this struct was generated from the following file:

- Scene.h

3.14 Timer Class Reference

Public Member Functions

- constexpr **Timer** (const Timer &)=delete
- constexpr **Timer** (Timer &&)=delete

Static Public Member Functions

- static double MeasureInMillis (std::function< void()> f)

3.14.1 Detailed Description

Definition at line 7 of file Timer.h.

3.14.2 Member Function Documentation

3.14.2.1 MeasureInMillis()

```
static double Timer::MeasureInMillis (
            std::function< void()> f )  [inline], [static]
```

Definition at line 13 of file Timer.h.

```
00013                                                          {
00014         auto t1 = std::chrono::high_resolution_clock::now();
00015         f();
00016         auto t2 = std::chrono::high_resolution_clock::now();
00017         static_cast<std::chrono::duration<float, std::milli>>(t2 - t1).count();
00018         return std::chrono::duration_cast<std::chrono::duration<float, std::milli»(t2 - t1).count();
00019     }
```

The documentation for this class was generated from the following file:

- Timer.h

Chapter 4

File Documentation

4.1 AABB.h

```
#ifndef _AABB_H
#define _AABB_H

#include "math/Math.h"

struct AABB {
    Math::Vector3f min, max;

    constexpr AABB() noexcept = default;

    constexpr AABB(const Math::Vector3f &a, const Math::Vector3f &b) noexcept :
        min(Math::Min(a, b)), max(Math::Max(a, b)) {}

    constexpr AABB(const AABB &other) noexcept = default;

    constexpr AABB(const AABB &a, const AABB &b) noexcept :
        min(Math::Min(a.min, b.min)), max(Math::Max(a.max, b.max)) {}

    constexpr float GetSurfaceArea() const noexcept {
        float a, b, c;
        a = max.x - min.x;
        b = max.y - min.y;
        c = max.z - min.z;
        return 2.f * a * b + 2.f * b * c + 2.f * a * c;
    }

    constexpr bool IntersectsRay(const Ray &ray, float t_min, float t_max) const noexcept {
        auto one_over_direction = 1.f / ray.direction;
        auto origin = ray.origin;

        auto t0 = (min - origin) * one_over_direction;
        auto t1 = (max - origin) * one_over_direction;

        if (one_over_direction.x < 0.f) {
            std::swap(t0.x, t1.x);
        }

        if (one_over_direction.y < 0.f) {
            std::swap(t0.y, t1.y);
        }

        if (one_over_direction.z < 0.f) {
            std::swap(t0.z, t1.z);
        }

        t_min = Math::Max(t_min, Math::Max(Math::Max(t0.x, t0.y), t0.z));
        t_max = Math::Min(t_max, Math::Min(Math::Min(t1.x, t1.y), t1.z));

        if (t_max <= t_min) {
            return false;
        }

        return true;
    }

    constexpr static AABB Empty() noexcept {
        AABB aabb;
        aabb.min = Math::Vector3f(+Math::Constants::Infinity<float>);
        aabb.max = Math::Vector3f(-Math::Constants::Infinity<float>);

        return aabb;
    }
};

#endif
```

4.2 AccelerationStructure.cpp

```
#include "AccelerationStructure.h"
#include "Utilities.hpp"
#include "hittable/NonHittable.h"

AccelerationStructure::AccelerationStructure() noexcept {
    m_Root = new BVHNode(new NonHittable());
}

AccelerationStructure::AccelerationStructure(std::span<HittableObjectPtr> objects) noexcept {
    Update(objects);
}

AccelerationStructure::~AccelerationStructure() noexcept {
    if (m_Root != nullptr) {
        BVHNode::FreeMemory(m_Root);
    }
}

void AccelerationStructure::Update(std::span<HittableObjectPtr> objects) noexcept {
    if (m_Root != nullptr) {
        BVHNode::FreeMemory(m_Root);
    }

    if (objects.empty()) {
        m_Root = new BVHNode(new NonHittable());
    } else {
        m_Root = BVHNode::MakeHierarchySAH(objects, 0, static_cast<int>(objects.size()));
    }
}

void AccelerationStructure::Hit(const Ray &ray, float tMin, float tMax, HitPayload &payload) const noexcept {
    m_Root->Hit(ray, tMin, tMax, payload);
}
```

4.3 AccelerationStructure.h

```
#ifndef _ACCELERATION_STRUCTURE_H
#define _ACCELERATION_STRUCTURE_H

#include "hittable/HittableObject.h"
#include "math/Math.h"
#include "AABB.h"
#include "Ray.h"
#include "HitPayload.h"
#include "BVHNode.h"

#include <span>

class AccelerationStructure {
public:
    AccelerationStructure() noexcept;

    AccelerationStructure(std::span<HittableObjectPtr> objects) noexcept;

    ~AccelerationStructure() noexcept;

    void Update(std::span<HittableObjectPtr> objects) noexcept;

    void Hit(const Ray &ray, float tMin, float tMax, HitPayload &payload) const noexcept;

private:
    BVHNode *m_Root = nullptr;
};

#endif
```

4.4 Application.cpp

```cpp
#include "Application.h"
#include "Utilities.hpp"
#include "Timer.h"

#include "../imgui-docking/imgui.h"
#include "../imgui-docking/backends/imgui_impl_glfw.h"
#include "../imgui-docking/backends/imgui_impl_opengl3.h"

#define STB_IMAGE_IMPLEMENTATION
#include "../stb-master/stb_image.h"

#define GL_SILENCE_DEPRECATION
#if defined(IMGUI_IMPL_OPENGL_ES2)
#include <GLES2/gl2.h>
#endif

#include <iostream>
#include <chrono>
#include <fstream>

Application::Application(int windowWidth, int windowHeight) noexcept :
    m_InitialWindowWidth(windowWidth), m_InitialWindowHeight(windowHeight),
    m_LastViewportWidth(-1), m_LastViewportHeight(-1),
    m_Renderer(windowWidth, windowHeight),
    m_TotalRenderTime(0.f), m_LastRenderTime(0.f) {

    m_SaveImageFilePath = new char[c_AnyInputFilePathLength];
    m_SceneFilePath = new char[c_AnyInputFilePathLength];
    m_ModelFilePath = new char[c_AnyInputFilePathLength];
    m_MaterialDirectory = new char[c_AnyInputFilePathLength];
    memset(m_SaveImageFilePath, 0, sizeof(m_SaveImageFilePath));
    memset(m_SceneFilePath, 0, sizeof(m_SceneFilePath));
    memset(m_ModelFilePath, 0, sizeof(m_ModelFilePath));
    memset(m_MaterialDirectory, 0, sizeof(m_MaterialDirectory));

    m_AddMaterial.albedo = {0.f, 0.f, 0.f};
    m_AddMaterial.metallic = 0.f;
    m_AddMaterial.specular = 0.f;
    m_AddMaterial.roughness = 0.f;
    m_AddMaterial.emissionPower = 0.f;
    m_AddMaterial.index = -1;

    m_AddSphere = Shapes::Sphere(Math::Vector3f(0.f), 0.f, nullptr);
    m_AddTriangle = Shapes::Triangle(Math::Vector3f(0.f), Math::Vector3f(0.f), Math::Vector3f(0.f), nullptr);
    m_AddBox = Shapes::Box(Math::Vector3f(0.f), Math::Vector3f(0.f), nullptr);

    m_AddSphereMaterialIndex = -1;
    m_AddTriangleMaterialIndex = -1;
    m_AddBoxMaterialIndex = -1;

    m_RayMissColor = Math::Vector3f(0.f);

    m_Scene.camera = Camera(windowWidth, windowHeight);

    LoadSceneFromFile(c_DefaultScenePath);
}

int Application::Run() noexcept {
    if (!glfwInit()) {
        std::cerr « "Failed to initialize GLFW\n";
        return -1;
    }

#if defined(IMGUI_IMPL_OPENGL_ES2)
    const char* glsl_version = "#version 100";
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
    glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_ES_API);
#elif defined(__APPLE__)
    const char* glsl_version = "#version 150";
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#else
    const char* glsl_version = "#version 130";
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
#endif

    m_Window = glfwCreateWindow(m_InitialWindowWidth, m_InitialWindowHeight, c_WindowTitle, nullptr, nullptr);
    if (m_Window == nullptr) {
        std::cerr « "Failed to create window\n";
        return -1;
    }

    GLFWimage *icon = new GLFWimage();
    icon->pixels = stbi_load("icon/icon.png", &icon->width, &icon->height, 0, 4);
    glfwSetWindowIcon(m_Window, 1, icon);
    stbi_image_free(icon->pixels);
    delete icon;

    glfwMakeContextCurrent(m_Window);

    ImGui::CreateContext();
    ImGuiIO &io = ImGui::GetIO();
    io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;

    UpdateThemeStyle();

    ImGui_ImplGlfw_InitForOpenGL(m_Window, true);
    ImGui_ImplOpenGL3_Init(glsl_version);

    MainLoop();

    ImGui_ImplOpenGL3_Shutdown();
    ImGui_ImplGlfw_Shutdown();
    ImGui::DestroyContext();

    glfwDestroyWindow(m_Window);
    glfwTerminate();

    return 0;
}

void Application::MainLoop() noexcept {
    while (!glfwWindowShouldClose(m_Window)) {
        glfwPollEvents();

        ImGui_ImplOpenGL3_NewFrame();
        ImGui_ImplGlfw_NewFrame();
        ImGui::NewFrame();

        const ImGuiViewport* viewport = ImGui::GetMainViewport();

        ImGui::SetNextWindowPos(viewport->WorkPos);
        ImGui::SetNextWindowSize(viewport->WorkSize);
        ImGui::SetNextWindowViewport(viewport->ID);

        ImGuiWindowFlags dockWindowFlags = ImGuiWindowFlags_NoDocking;
        dockWindowFlags |= ImGuiWindowFlags_NoTitleBar | ImGuiWindowFlags_NoCollapse |
ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoMove;
        dockWindowFlags |= ImGuiWindowFlags_NoBringToFrontOnFocus | ImGuiWindowFlags_NoNavFocus;

        {
            OnUpdate();
        }

        ImGui::Render();
        ImGui::UpdatePlatformWindows();

        int display_w, display_h;
        glfwGetFramebufferSize(m_Window, &display_w, &display_h);
        glViewport(0, 0, display_w, display_h);
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);

        ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

        glfwSwapBuffers(m_Window);
    }
}

void Application::OnUpdate() noexcept {
    const ImGuiViewport* viewport = ImGui::GetMainViewport();
    ImGuiWindowFlags frameFlags = ImGuiViewportFlags_IsPlatformWindow
                                | ImGuiViewportFlags_NoDecoration
                                | ImGuiViewportFlags_NoTaskBarIcon
                                | ImGuiViewportFlags_NoAutoMerge
                                | ImGuiWindowFlags_NoCollapse
                                | ImGuiWindowFlags_NoResize;

    ImGui::SetNextWindowPos(viewport->WorkPos);
    ImGui::SetNextWindowSize({viewport->WorkSize.x * 0.7f, viewport->WorkSize.y});
    ImGui::SetNextWindowViewport(viewport->ID);

    ImGui::Begin("Frame", nullptr, frameFlags);
    {
        int viewportWidth = (int)ImGui::GetContentRegionAvail().x;
        int viewportHeight = (int)ImGui::GetContentRegionAvail().y;

        int width = -1, height = -1;
        glfwGetWindowSize(m_Window, &width, &height);

        if ((m_LastViewportWidth != viewportWidth || m_LastViewportHeight != viewportHeight) && width * height > 0) {
            m_LastViewportWidth = viewportWidth;
            m_LastViewportHeight = viewportHeight;

            m_Scene.camera.OnViewportResize(m_LastViewportWidth, m_LastViewportHeight);

            m_Renderer.OnResize(m_LastViewportWidth, m_LastViewportHeight);
        }

        Image *image = m_Renderer.GetImage();
        if (image != nullptr) {
            ImGui::Image((void*)(intptr_t)image->GetDescriptor(), ImGui::GetContentRegionAvail());
        }
    }
    ImGui::End();

    ImGui::SetNextWindowPos({viewport->WorkPos.x + viewport->WorkSize.x * 0.7f, viewport->WorkPos.y});
    ImGui::SetNextWindowSize({viewport->WorkSize.x * 0.3f, viewport->WorkSize.y});
    ImGui::SetNextWindowViewport(viewport->ID);

    ImGui::Begin("Options", nullptr, ImGuiWindowFlags_NoTitleBar);
    {
        m_LastID = 0;
        m_SomeObjectChanged = false;
        m_SomeGeometryChanged = false;

        ProcessSceneCollapsingHeaders();

        ImGui::Checkbox("Accumulate", Math::ValuePointer(m_Renderer.Accumulate()));
        ImGui::Checkbox("Accelerate", Math::ValuePointer(m_Renderer.Accelerate()));
        if (ImGui::InputInt("Used threads", Math::ValuePointer(m_Renderer.UsedThreadCount()))) {
            m_Renderer.SetUsedThreadCount(m_Renderer.UsedThreadCount());
        }
        ImGui::InputInt("Ray depth", Math::ValuePointer(m_Renderer.RayDepth()));
        ImGui::InputFloat("Gamma", Math::ValuePointer(m_Renderer.Gamma()));

        if (ImGui::ColorEdit3("Ray miss color", Math::ValuePointer(m_RayMissColor))) {
            m_Renderer.OnRayMiss([this](const Ray&){ return m_RayMissColor; });
        }

        if (ImGui::Button("Reset", {viewport->WorkSize.x * 0.05f, viewport->WorkSize.y * 0.1f}) || m_Renderer.Accumulate()) {
            m_Scene.camera.ComputeRayDirections();

            m_LastRenderTime = Timer::MeasureInMillis([this](){
                if (m_Renderer.Accelerate()) {
                    m_Renderer.Render(m_Scene.camera, m_AccelerationStructure, m_Lights, m_Scene.materials);
                } else {
                    m_Renderer.Render(m_Scene.camera, m_Objects, m_Lights, m_Scene.materials);
                }
            });

            if (m_Renderer.Accumulate()) {
                m_TotalRenderTime += m_LastRenderTime;
            } else {
                m_TotalRenderTime = 0.f;
            }
        }

        if (ImGui::Checkbox("Dark theme", Math::ValuePointer(m_DarkTheme))) {
            UpdateThemeStyle();
        }

        ImGui::InputText("##save_image", m_SaveImageFilePath, c_AnyInputFilePathLength);
        ImGui::SameLine();
        if (ImGui::Button("Save image")) {
            m_Renderer.SaveImage(m_SaveImageFilePath);
        }

        ImGui::InputText("##save_scene", m_SceneFilePath, c_AnyInputFilePathLength);
        if (ImGui::Button("Save scene")) {
            SaveSceneToFile(m_SceneFilePath);
        }

        ImGui::SameLine();
        if (ImGui::Button("Load scene")) {
            LoadSceneFromFile(m_SceneFilePath);
        }

        ImGui::Text("Last render time: %fms", m_LastRenderTime);
        ImGui::Text("Average render time: %fms", m_TotalRenderTime /
(Math::Max(m_Renderer.GetFrameIndex() - 1, 1)));
        ImGui::Text("Accumulated frame count: %d", Math::Max(m_Renderer.GetFrameIndex() - 1, 1));
    }

    ImGui::End();
}

void Application::UpdateThemeStyle() noexcept {
    if (m_DarkTheme) {
        ImGui::StyleColorsDark();
    } else {
        ImGui::StyleColorsLight();
    }
}

void Application::ProcessSceneCollapsingHeaders() noexcept {
    ProcessCameraCollapsingHeader();
    ProcessSpheresCollapsingHeader();
    ProcessTrianglesCollapsingHeader();
    ProcessBoxesCollapsingHeader();
    ProcessModelsCollapsingHeader();
    ProcessMaterialsCollapsingHeader();

    if (m_SomeObjectChanged) {
        UpdateObjects();
    }

    if (m_SomeGeometryChanged) {
        m_AccelerationStructure.Update(m_Objects);
    }
}

void Application::ProcessCameraCollapsingHeader() noexcept {
    ImGui::PushID(m_LastID++);

    if (ImGui::CollapsingHeader("Camera", nullptr)) {
        bool cameraNeedUpdate = false;
        ImGui::InputFloat3("Position", Math::ValuePointer(m_Scene.camera.Position()));
        ImGui::InputFloat3("Target", Math::ValuePointer(m_Scene.camera.Target()));
        ImGui::InputFloat("Vertical FOV", Math::ValuePointer(m_Scene.camera.VerticalFovInDegrees()));
        ImGui::InputFloat3("Up", Math::ValuePointer(m_Scene.camera.Up()));
    }

    ImGui::PopID();
}

void Application::ProcessSpheresCollapsingHeader() noexcept {
    int deleteIndex = -1;
    if (ImGui::CollapsingHeader("Spheres", nullptr)) {
        for (int i = 0; i < (int)m_Scene.spheres.size(); ++i) {
            ImGui::PushID(m_LastID++);

            Shapes::Sphere &sphere = m_Scene.spheres[i];
            ImGui::Text("Sphere %d:", i);

            if (ImGui::Button("Delete")) {
                deleteIndex = i;
            }

            if (ImGui::InputFloat("Radius", Math::ValuePointer(sphere.radius))) {
                sphere = Shapes::Sphere(sphere.center, sphere.radius, sphere.material);
                m_SomeGeometryChanged = true;
            }

            if (ImGui::InputFloat3("Position", Math::ValuePointer(sphere.center))) {
                m_SomeGeometryChanged = true;
            }

            if (ImGui::InputInt("Material index", Math::ValuePointer(m_SphereMaterialIndices[i]))) {
                sphere.material = &m_Scene.materials[m_SphereMaterialIndices[i]];
            }

            ImGui::PopID();
        }

        ImGui::PushID(m_LastID++);

        if (ImGui::Button("Add")) {
            m_Scene.spheres.push_back(m_AddSphere);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }

        if (ImGui::InputFloat("Radius", Math::ValuePointer(m_AddSphere.radius))) {
            m_AddSphere = Shapes::Sphere(m_AddSphere.center, m_AddSphere.radius,
m_AddSphere.material);
        }

        ImGui::InputFloat3("Position", Math::ValuePointer(m_AddSphere.center));

        if (ImGui::InputInt("Material index", Math::ValuePointer(m_AddSphereMaterialIndex))) {
            m_AddSphere.material = &m_Scene.materials[m_AddSphereMaterialIndex];
        }

        ImGui::PopID();

        if (deleteIndex >= 0) {
            m_Scene.spheres.erase(m_Scene.spheres.cbegin() + deleteIndex);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }
    }
}

void Application::ProcessTrianglesCollapsingHeader() noexcept {
    int deleteIndex = -1;
    if (ImGui::CollapsingHeader("Triangles", nullptr)) {
        for (int i = 0; i < (int)m_Scene.triangles.size(); ++i) {
            ImGui::PushID(m_LastID++);

            Shapes::Triangle &triangle = m_Scene.triangles[i];
            ImGui::Text("Triangle %d:", i);

            if (ImGui::Button("Delete")) {
                deleteIndex = i;
            }

            if (ImGui::InputFloat3("Vertex 0", Math::ValuePointer(triangle.vertices[0]))) {
                triangle = Shapes::Triangle(triangle.vertices[0], triangle.vertices[1], triangle.vertices[2], triangle.material);
                m_SomeGeometryChanged = true;
            }
            if (ImGui::InputFloat3("Vertex 1", Math::ValuePointer(triangle.vertices[1]))) {
                triangle = Shapes::Triangle(triangle.vertices[0], triangle.vertices[1], triangle.vertices[2], triangle.material);
                m_SomeGeometryChanged = true;
            }
            if (ImGui::InputFloat3("Vertex 2", Math::ValuePointer(triangle.vertices[2]))) {
                triangle = Shapes::Triangle(triangle.vertices[0], triangle.vertices[1], triangle.vertices[2], triangle.material);
                m_SomeGeometryChanged = true;
            }

            if (ImGui::InputInt("Material index", Math::ValuePointer(m_TriangleMaterialIndices[i]))) {
                triangle.material = &m_Scene.materials[m_TriangleMaterialIndices[i]];
            }

            ImGui::PopID();
        }

        ImGui::PushID(m_LastID++);

        if (ImGui::Button("Add")) {
            m_Scene.triangles.push_back(m_AddTriangle);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }

        if (ImGui::InputFloat3("Vertex 0", Math::ValuePointer(m_AddTriangle.vertices[0]))) {
            m_AddTriangle = Shapes::Triangle(m_AddTriangle.vertices[0], m_AddTriangle.vertices[1], m_AddTriangle.vertices[2], m_AddTriangle.material);
        }
        if (ImGui::InputFloat3("Vertex 1", Math::ValuePointer(m_AddTriangle.vertices[1]))) {
            m_AddTriangle = Shapes::Triangle(m_AddTriangle.vertices[0], m_AddTriangle.vertices[1], m_AddTriangle.vertices[2], m_AddTriangle.material);
        }
        if (ImGui::InputFloat3("Vertex 2", Math::ValuePointer(m_AddTriangle.vertices[2]))) {
            m_AddTriangle = Shapes::Triangle(m_AddTriangle.vertices[0], m_AddTriangle.vertices[1], m_AddTriangle.vertices[2], m_AddTriangle.material);
        }

        if (ImGui::InputInt("Material index", Math::ValuePointer(m_AddTriangleMaterialIndex))) {
            m_AddTriangle.material = &m_Scene.materials[m_AddTriangleMaterialIndex];
        }

        ImGui::PopID();

        if (deleteIndex >= 0) {
            m_Scene.triangles.erase(m_Scene.triangles.cbegin() + deleteIndex);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }
    }
}
```

```

void Application::ProcessBoxesCollapsingHeader() noexcept {
    int deleteIndex = -1;
    if (ImGui::CollapsingHeader("Boxes", nullptr)) {
        for (int i = 0; i < (int)m_Scene.boxes.size(); ++i) {
            ImGui::PushID(m_LastID++);

            ImGui::Text("Box: %d", i);

            if (ImGui::Button("Delete")) {
                deleteIndex = i;
            }

            Shapes::Box &box = m_Scene.boxes[i];

            if (ImGui::InputFloat3("First corner", Math::ValuePointer(box.min))) {
                box = Shapes::Box(box.min, box.max, box.material);
                m_SomeGeometryChanged = true;
            }

            if (ImGui::InputFloat3("Second corner", Math::ValuePointer(box.max))) {
                box = Shapes::Box(box.min, box.max, box.material);
                m_SomeGeometryChanged = true;
            }

            if (ImGui::InputInt("Material index", Math::ValuePointer(m_BoxMaterialIndices[i]))) {
                box = Shapes::Box(box.min, box.max, &m_Scene.materials[m_BoxMaterialIndices[i]]);
            }

            ImGui::PopID();
        }

        ImGui::PushID(m_LastID++);

        if (ImGui::Button("Add")) {
            m_Scene.boxes.push_back(m_AddBox);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }

        if (ImGui::InputFloat3("First corner", Math::ValuePointer(m_AddBox.min))) {
            m_AddBox = Shapes::Box(m_AddBox.min, m_AddBox.max, m_AddBox.material);
        }
        if (ImGui::InputFloat3("Second corner", Math::ValuePointer(m_AddBox.max))) {
            m_AddBox = Shapes::Box(m_AddBox.min, m_AddBox.max, m_AddBox.material);
        }

        if (ImGui::InputInt("Material index", Math::ValuePointer(m_AddBoxMaterialIndex))) {
            m_AddBox.material = &m_Scene.materials[m_AddBoxMaterialIndex];
        }

        ImGui::PopID();

        if (deleteIndex >= 0) {
            m_Scene.boxes.erase(m_Scene.boxes.cbegin() + deleteIndex);
            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }
    }
}

void Application::ProcessModelsCollapsingHeader() noexcept {
    int deleteIndex = -1;
    if (ImGui::CollapsingHeader("Models", nullptr)) {
        for (int i = 0; i < (int)m_Scene.models.size(); ++i) {
            ImGui::PushID(m_LastID++);

            auto model = m_Scene.models[i];

            ImGui::Text("Model: %d", i);

            auto materials = model->GetMaterials();
            for (int i = 0; i < (int)materials.size(); ++i) {
                ImGui::PushID(m_LastID++);

                Material &material = materials[i];
                ImGui::Text("Material %d:", material.index);

                ImGui::ColorEdit3("Albedo", Math::ValuePointer(material.albedo));
                ImGui::InputFloat("Emission power", Math::ValuePointer(material.emissionPower));
                ImGui::InputFloat("Metallic", Math::ValuePointer(material.metallic));
                ImGui::InputFloat("Roughness", Math::ValuePointer(material.roughness));
                ImGui::InputFloat("Specular", Math::ValuePointer(material.specular));

                ImGui::PopID();
            }

            if (ImGui::Button("Delete")) {
                deleteIndex = i;
            }

            ImGui::PopID();
        }

        ImGui::PushID(m_LastID++);

        if (ImGui::Button("Import")) {
            auto result = Model::LoadOBJ(m_ModelFilePath, m_MaterialDirectory);

            if (!result.warning.empty()) {
                std::cout « "Warnings occured while loading model " « m_ModelFilePath « " with material directory: " « m_MaterialDirectory « ": " « result.warning « ’\n’;
            }

            if (result.IsFailure()) {
                std::cerr « "Failed to import model " « m_ModelFilePath « " with material directory: " « m_MaterialDirectory « ’\n’;
            } else {
                std::cout « "Loaded model " « m_ModelFilePath « " with material directory: " « m_MaterialDirectory « ’\n’;

                m_Scene.models.push_back(result.model);
                m_SomeObjectChanged = true;
                m_SomeGeometryChanged = true;
            }
        }

        ImGui::InputText("Path (.obj)", m_ModelFilePath, c_AnyInputFilePathLength);
        ImGui::InputText("Material folder", m_MaterialDirectory, c_AnyInputFilePathLength);

        ImGui::PopID();

        if (deleteIndex >= 0) {
            delete m_Scene.models[deleteIndex];
            m_Scene.models.erase(m_Scene.models.cbegin() + deleteIndex);

            m_SomeObjectChanged = true;
            m_SomeGeometryChanged = true;
        }
    }
}

void Application::ProcessMaterialsCollapsingHeader() noexcept {
    int deleteIndex = -1;
    if (ImGui::CollapsingHeader("Materials", nullptr)) {
        for (int i = 0; i < (int)m_Scene.materials.size(); ++i) {
            ImGui::PushID(m_LastID++);

            Material &material = m_Scene.materials[i];
            ImGui::Text("Material %d:", material.index);

            if (ImGui::Button("Delete")) {
                deleteIndex = i;
            }

            ImGui::ColorEdit3("Albedo", Math::ValuePointer(material.albedo));
            ImGui::InputFloat("Emission power", Math::ValuePointer(material.emissionPower));
            ImGui::InputFloat("Metallic", Math::ValuePointer(material.metallic));
            ImGui::InputFloat("Roughness", Math::ValuePointer(material.roughness));
            ImGui::InputFloat("Specular", Math::ValuePointer(material.specular));

            ImGui::PopID();
        }

        ImGui::PushID(m_LastID++);

        if (ImGui::Button("Add")) {
            int maxIndex = -1;
            for (const auto &material : m_Scene.materials) {
                maxIndex = Math::Max(maxIndex, material.index);
            }

            m_AddMaterial.index = maxIndex + 1;
            m_Scene.materials.push_back(m_AddMaterial);
            UpdateObjectMaterials();
        }

        ImGui::ColorEdit3("Albedo", Math::ValuePointer(m_AddMaterial.albedo));
        ImGui::InputFloat("Emission power", Math::ValuePointer(m_AddMaterial.emissionPower));
        ImGui::InputFloat("Metallic", Math::ValuePointer(m_AddMaterial.metallic));
        ImGui::InputFloat("Roughness", Math::ValuePointer(m_AddMaterial.roughness));
        ImGui::InputFloat("Specular", Math::ValuePointer(m_AddMaterial.specular));

        ImGui::PopID();

        if (deleteIndex >= 0) {
            m_Scene.materials.erase(m_Scene.materials.cbegin() + deleteIndex);
        }
    }
}

void Application::LoadSceneFromFile(const std::filesystem::path &pathToFile) noexcept {
    std::ifstream fileStream(pathToFile, std::ios::binary);
    if (!fileStream) {
        std::cerr « "Failed to open file: " « pathToFile « ’\n’;
        return;
    }

    auto error = m_Scene.Deserialize(fileStream);
    if (error.has_value()) {
        std::cerr « "Failed to deserialize scene: " « pathToFile « ’\n’;
        return;
    } else {
        std::cout « "Loaded scene: " « pathToFile « ’\n’;
    }

    UpdateObjects();
    UpdateLights();
    m_AccelerationStructure.Update(m_Objects);
}

void Application::SaveSceneToFile(const std::filesystem::path &pathToFile) const noexcept {
    std::ofstream fileStream(pathToFile, std::ios::binary);
    if (!fileStream) {
        std::cerr « "Failed to open file: " « pathToFile « ’\n’;
        return;
    }

    auto error = m_Scene.Serialize(fileStream);
    if (error.has_value()) {
        std::cerr « "Failed to serialize scene: " « pathToFile « ’\n’;
    }
}

void Application::UpdateObjects() noexcept {
    m_Objects.clear();

    m_SphereMaterialIndices.clear();
    for (auto &sphere : m_Scene.spheres) {
        m_Objects.push_back(&sphere);
        m_SphereMaterialIndices.push_back(sphere.material->index);
    }

    m_TriangleMaterialIndices.clear();
    for (auto &triangle : m_Scene.triangles) {
        m_Objects.push_back(&triangle);
        m_TriangleMaterialIndices.push_back(triangle.material->index);
    }

    m_BoxMaterialIndices.clear();
    for (auto &box : m_Scene.boxes) {
        m_Objects.push_back(&box);
        m_BoxMaterialIndices.push_back(box.material->index);
    }

    for (auto model : m_Scene.models) {
        m_Objects.push_back(model);
    }
}

void Application::UpdateLights() noexcept {
    m_Lights.clear();

    for (auto &sphere : m_Scene.spheres) {
        if (sphere.material->emissionPower > 0.f) {
            m_Lights.emplace_back(&sphere, sphere.material->GetEmission());
        }
    }

    for (auto &triangle : m_Scene.triangles) {
        if (triangle.material->emissionPower > 0.f) {
            m_Lights.emplace_back(&triangle, triangle.material->GetEmission());
        }
    }

    for (auto &box : m_Scene.boxes) {
        if (box.material->emissionPower > 0.f) {
            m_Lights.emplace_back(&box, box.material->GetEmission());
        }
    }

    for (auto model : m_Scene.models) {
        auto lights = model->GetLightSources();
        m_Lights.insert(m_Lights.cend(), lights.begin(), lights.end());
    }
}

void Application::UpdateObjectMaterials() noexcept {
    int sphereCount = static_cast<int>(m_Scene.spheres.size());
    for (int i = 0; i < sphereCount; ++i) {
        m_Scene.spheres[i].material = &*std::find_if(m_Scene.materials.cbegin(), m_Scene.materials.cend(), [this, i](const auto &material) {
            return material.index == m_SphereMaterialIndices[i];
        });
    }

    int triangleCount = static_cast<int>(m_Scene.triangles.size());
    for (int i = 0; i < triangleCount; ++i) {
        m_Scene.triangles[i].material = &*std::find_if(m_Scene.materials.cbegin(), m_Scene.materials.cend(), [this, i](const auto &material) {
            return material.index == m_TriangleMaterialIndices[i];
        });
    }

    int boxCount = static_cast<int>(m_Scene.boxes.size());
    for (int i = 0; i < boxCount; ++i) {
        m_Scene.boxes[i].material = &*std::find_if(m_Scene.materials.cbegin(), m_Scene.materials.cend(), [this, i](const auto &material) {
            return material.index == m_BoxMaterialIndices[i];
        });
    }
}
```

4.5 Application.h

```
#ifndef _APPLICATION_H
#define _APPLICATION_H

#include "../glfw-3.4/include/GLFW/glfw3.h"

#include "Scene.h"
#include "Camera.h"
#include "AccelerationStructure.h"
#include "Renderer.h"

#include <cstring>
#include <filesystem>

class Application {
public:
    Application(int windowWidth, int windowHeight) noexcept;

    int Run() noexcept;

private:
    void MainLoop() noexcept;

    void OnUpdate() noexcept;

    void ProcessSceneCollapsingHeaders() noexcept;

    void ProcessCameraCollapsingHeader() noexcept;

    void ProcessSpheresCollapsingHeader() noexcept;

    void ProcessTrianglesCollapsingHeader() noexcept;

    void ProcessBoxesCollapsingHeader() noexcept;

    void ProcessModelsCollapsingHeader() noexcept;

    void ProcessMaterialsCollapsingHeader() noexcept;

    void UpdateThemeStyle() noexcept;

    void LoadSceneFromFile(const std::filesystem::path &pathToFile) noexcept;

    void SaveSceneToFile(const std::filesystem::path &pathToFile) const noexcept;

    void UpdateObjects() noexcept;

    void UpdateLights() noexcept;

    void UpdateObjectMaterials() noexcept;

private:
    int m_InitialWindowWidth, m_InitialWindowHeight;
    int m_LastViewportWidth, m_LastViewportHeight;
    GLFWwindow *m_Window = nullptr;
    bool m_DarkTheme = false;

    int m_LastID;
    bool m_SomeObjectChanged;
    bool m_SomeGeometryChanged;

    double m_TotalRenderTime;
    double m_LastRenderTime;

    char *m_SaveImageFilePath;
    char *m_SceneFilePath;

    Scene m_Scene;
    AccelerationStructure m_AccelerationStructure;
    Renderer m_Renderer;

    Math::Vector3f m_RayMissColor;

    std::vector<int> m_SphereMaterialIndices;
    std::vector<int> m_TriangleMaterialIndices;
    std::vector<int> m_BoxMaterialIndices;

    std::vector<HittableObject*> m_Objects;
    std::vector<Light> m_Lights;

    Material m_AddMaterial;
    Shapes::Sphere m_AddSphere;
    Shapes::Triangle m_AddTriangle;
    Shapes::Box m_AddBox;
    char *m_ModelFilePath;
    char *m_MaterialDirectory;

    int m_AddSphereMaterialIndex;
    int m_AddTriangleMaterialIndex;
    int m_AddBoxMaterialIndex;

    constexpr static const char *c_WindowTitle = "Path Tracing";
    constexpr static const char *c_DefaultScenePath = "assets/dft2.scn";
    constexpr static int c_AnyInputFilePathLength = 128;
};

#endif
```

4.6 BSDF.cpp

```
#include "BSDF.h"
#include "Utilities.hpp"

Math::Vector3f BSDF::Sample(const Ray &ray, const HitPayload &payload, Math::Vector3f &throughput) noexcept {
    float diffuseRatio = 0.5f * (1.f - m_Material->metallic);
    float specularRatio = 1.f - diffuseRatio;

    Math::Vector3f V = -ray.direction;

    Math::Vector3f reflectionDirection;
    if (Utilities::RandomFloatInZeroToOne() < diffuseRatio) {
        reflectionDirection = Utilities::RandomInHemisphere(payload.normal);
    } else {
        Math::Vector3f halfVec;
        {
            Math::Vector2f Xi{Utilities::RandomFloatInZeroToOne(), Utilities::RandomFloatInZeroToOne()};
            Math::Vector3f N = payload.normal;

            float a = m_Material->roughness * m_Material->roughness;

            float phi = Math::Constants::Tau<float> * Xi.x;
            float cosTheta = Math::Sqrt((1.f - Xi.y) / (1.f + (a * a - 1.f) * Xi.y));
            float sinTheta = Math::Sqrt(1.f - cosTheta * cosTheta);

            Math::Vector3f H;
            H.x = Math::Cos(phi) * sinTheta;
            H.y = Math::Sin(phi) * sinTheta;
            H.z = cosTheta;

            Math::Vector3f up = Math::Abs(N.z) < 0.999f ? Math::Vector3f(0.0, 0.0, 1.0) : Math::Vector3f(1.0, 0.0, 0.0);
            Math::Vector3f tangent = Math::Normalize(Math::Cross(up, N));
            Math::Vector3f bitangent = Math::Cross(N, tangent);

            halfVec = tangent * H.x + bitangent * H.y + N * H.z;
            halfVec = Math::Normalize(halfVec);
        }

        reflectionDirection = Math::Normalize(2.f * Math::Dot(V, halfVec) * halfVec - V);
    }

    auto DistributionGGX = [](const Math::Vector3f &N, const Math::Vector3f &H, float roughness) {
        float a = roughness * roughness;
        float a2 = a * a;
        float NdotH = Math::Max(Math::Dot(N, H), 0.f);
        float NdotH2 = NdotH * NdotH;

        float nom = a2;
        float denom = (NdotH2 * (a2 - 1.f) + 1.f);
        denom = Math::Constants::Pi<float> * denom * denom;

        return nom / denom;
    };

    auto GeometrySchlickGGX = [](float NdotV, float roughness) {
        float r = (roughness + 1.f);
        float k = (r * r) / 8.f;

        float nom = NdotV;
        float denom = NdotV * (1.f - k) + k;

        return nom / denom;
    };

    auto GeometrySmith = [&](const Math::Vector3f &N, const Math::Vector3f &V, const Math::Vector3f &L, float roughness) {
        float NdotV = Math::Abs(Math::Dot(N, V));
        float NdotL = Math::Abs(Math::Dot(N, L));
        float ggx2 = GeometrySchlickGGX(NdotV, roughness);
        float ggx1 = GeometrySchlickGGX(NdotL, roughness);

        return ggx1 * ggx2;
    };

    auto FresnelSchlick = [](float cosTheta, const Math::Vector3f &F0) {
        return F0 + (1.f - F0) * Math::Pow(1.f - cosTheta, 5.f);
    };

    auto SpecularBRDF = [](float D, float G, const Math::Vector3f &F, const Math::Vector3f &V, const Math::Vector3f &L, const Math::Vector3f &N) {
        float NdotL = Math::Abs(Math::Dot(N, L));
        float NdotV = Math::Abs(Math::Dot(N, V));

        Math::Vector3f nominator = D * G * F;
        float denominator = 4.f * NdotV * NdotL + 0.001f;
        Math::Vector3f specularBrdf = nominator / denominator;

        return specularBrdf;
    };

    auto DiffuseBRDF = [](const Math::Vector3f &albedo) {
        return albedo * Math::Constants::InversePi<float>;
    };

    auto ImportanceSampleGGXPDF = [](float NDF, float NdotH, float VdotH) {
        return NDF * NdotH / (4.f * VdotH);
    };

    auto CosineSamplingPDF = [](float NdotL) {
        return NdotL * Math::Constants::InversePi<float>;
    };

    Math::Vector3f L = reflectionDirection;
    Math::Vector3f H = Math::Normalize(V + L);

    float NdotL = Math::Abs(Math::Dot(payload.normal, L));
    float NdotH = Math::Abs(Math::Dot(payload.normal, H));
    float VdotH = Math::Abs(Math::Dot(V, H));

    float NdotV = Math::Abs(Math::Dot(payload.normal, V));

    Math::Vector3f F0 = Math::Vector3f(0.08f, 0.08f, 0.08f);
    F0 = Math::Lerp(F0 * m_Material->specular, m_Material->albedo, m_Material->metallic);

    float NDF = DistributionGGX(payload.normal, H, m_Material->roughness);
    float G = GeometrySmith(payload.normal, V, L, m_Material->roughness);
    Math::Vector3f F = FresnelSchlick(Math::Max(Math::Dot(H, V), 0.f), F0);

    Math::Vector3f kS = F;
    Math::Vector3f kD = 1.f - kS;
    kD *= 1.0 - m_Material->metallic;

    Math::Vector3f specularBrdf = SpecularBRDF(NDF, G, F, V, L, payload.normal);

    float specularPdf = ImportanceSampleGGXPDF(NDF, NdotH, VdotH);

    Math::Vector3f diffuseBrdf = DiffuseBRDF(m_Material->albedo);
    float diffusePdf = CosineSamplingPDF(NdotL);

    Math::Vector3f totalBrdf = (diffuseBrdf * kD + specularBrdf) * NdotL;
    float totalPdf = diffuseRatio * diffusePdf + specularRatio * specularPdf;

    if (totalPdf > Math::Constants::Epsilon<float>) {
        throughput *= totalBrdf / totalPdf;
    }

    return reflectionDirection;
}
```

4.7 BSDF.h

```
#ifndef _BSDF_H
#define _BSDF_H

#include "Material.h"
#include "Ray.h"
#include "HitPayload.h"
#include "math/Math.h"

class BSDF {
public:
    constexpr BSDF(const Material *material) noexcept :
        m_Material(material) {}

    Math::Vector3f Sample(const Ray &ray, const HitPayload &payload, Math::Vector3f &throughput) noexcept;

private:
    const Material *m_Material;
};

#endif
```

4.8 BVHNode.h

```
#ifndef _BVHNODE_H
#define _BVHNODE_H

#include "Ray.h"
#include "AABB.h"
#include "HitPayload.h"
#include "hittable/HittableObject.h"

#include <span>
#include <vector>
#include <functional>

struct BVHNode {
    AABB aabb = AABB::Empty();
    BVHNode *left = nullptr, *right = nullptr;
    const HittableObject *object = nullptr;

    constexpr BVHNode() noexcept = default;

    constexpr BVHNode(BVHNode *left, BVHNode *right) noexcept :
        left(left), right(right), aabb(left->aabb, right->aabb) {}

    constexpr BVHNode(const HittableObject *object) noexcept :
        object(object), aabb(object->GetBoundingBox()) {}

    constexpr bool IsTerminating() const noexcept {
        return object != nullptr;
    }

    inline bool Hit(const Ray &ray, float tMin, float tMax, HitPayload &payload) noexcept {
        if (!aabb.IntersectsRay(ray, tMin, tMax)) {
            return false;
        }

        if (IsTerminating()) {
            return object->Hit(ray, tMin, tMax, payload);
        }

        bool anyHit = left->Hit(ray, tMin, tMax, payload);
        anyHit |= right->Hit(ray, tMin, Math::Min(tMax, payload.t), payload);

        return anyHit;
    }

    inline static BVHNode* MakeHierarchySAH(std::span<HittableObjectPtr> objects, int low, int high) noexcept {
        if (low + 1 == high) {
            return new BVHNode(objects[low]);
        }

        int n = high - low;
        std::vector<AABB> pref(n + 1);
        std::vector<AABB> suff(n + 1);

        float minValue = Math::Constants::Infinity<float>;
        int mid = -1;
        int dimension = -1;

        for (int d = 0; d < 3; ++d) {
            std::sort(objects.begin() + low, objects.begin() + high, c_ComparatorsSAH[d]);

            pref[0] = AABB::Empty();
            for (int i = 0; i < n; ++i) {
                pref[i + 1] = AABB(pref[i], objects[i + low]->GetBoundingBox());
            }

            suff[n] = AABB::Empty();
            for (int i = n - 1; i >= 0; --i) {
                suff[i] = AABB(objects[i + low]->GetBoundingBox(), suff[i + 1]);
            }

            float minValueAlongAxis = Math::Constants::Infinity<float>;
            int index = -1;
            for (int i = 0; i < n; ++i) {
                float value = pref[i + 1].GetSurfaceArea() * (float)(i + 1) + suff[i + 1].GetSurfaceArea() * (float)(n - i - 1);
                if (value < minValueAlongAxis) {
                    minValueAlongAxis = value;
                    index = i + low;
                }
            }

            if (minValueAlongAxis < minValue) {
                minValue = minValueAlongAxis;
                mid = index + 1;
                dimension = d;
            }
        }

        std::sort(objects.begin() + low, objects.begin() + high, c_ComparatorsSAH[dimension]);

        BVHNode *left = MakeHierarchySAH(objects, low, mid);
        BVHNode *right = MakeHierarchySAH(objects, mid, high);

        return new BVHNode(left, right);
    }

    inline static const std::function<bool(HittableObjectPtr, HittableObjectPtr)> c_ComparatorsSAH[3] = {
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().x < b->GetCentroid().x;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().y < b->GetCentroid().y;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetCentroid().z < b->GetCentroid().z;
        }
    };

    inline static BVHNode* MakeHierarchyNaive(std::span<HittableObjectPtr> objects, int low, int high) noexcept {
        AABB aabb = AABB::Empty();
        for (int i = low; i < high; ++i) {
            aabb = AABB(aabb, objects[i]->GetBoundingBox());
        }

        int longestAxisIndex = 0;
        float longestAxisLength = aabb.max.x - aabb.min.x;

        if (aabb.max.y - aabb.min.y > longestAxisLength) {
            longestAxisIndex = 1;
            longestAxisLength = aabb.max.y - aabb.min.y;
        }

        if (aabb.max.z - aabb.min.z > longestAxisLength) {
            longestAxisIndex = 2;
            longestAxisLength = aabb.max.z - aabb.min.z;
        }

        auto comparator = c_ComparatorsNaive[longestAxisIndex];

        if (low + 1 == high) {
            return new BVHNode(objects[low]);
        }

        std::sort(objects.begin() + low, objects.begin() + high, comparator);

        int mid = (low + high) / 2;
        BVHNode *left = MakeHierarchyNaive(objects, low, mid);
        BVHNode *right = MakeHierarchyNaive(objects, mid, high);

        return new BVHNode(left, right);
    }

    inline static const std::function<bool(HittableObjectPtr, HittableObjectPtr)>
c_ComparatorsNaive[3] = {
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.x < b->GetBoundingBox().min.x;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.y < b->GetBoundingBox().min.y;
        },
        [](HittableObjectPtr a, HittableObjectPtr b){
            return a->GetBoundingBox().min.z < b->GetBoundingBox().min.z;
        }
    };

    constexpr static void FreeMemory(BVHNode *node) noexcept {
        if (node == nullptr) {
            return;
        }

        FreeMemory(node->left);
        FreeMemory(node->right);

        delete node;
    }
};

#endif
```

4.9 Camera.cpp

```
#include "Camera.h"
#include "Utilities.hpp"

Camera::Camera(int viewportWidth, int viewportHeight, const Math::Vector3f &position, const
Math::Vector3f &target, float verticalFovInDegrees, const Math::Vector3f &up) noexcept :
    m_ViewportWidth(viewportWidth), m_ViewportHeight(viewportHeight), m_Position(position),
m_Target(target), m_VerticalFovInDegrees(verticalFovInDegrees), m_Up(up) {
    m_RayDirections.resize(m_ViewportWidth * m_ViewportHeight);
    ComputeRayDirections();
}

void Camera::OnViewportResize(int viewportWidth, int viewportHeight) noexcept {
    if (m_ViewportWidth == viewportWidth && m_ViewportHeight == viewportHeight) {
        return;
    }

    m_ViewportWidth = viewportWidth;
    m_ViewportHeight = viewportHeight;

    m_RayDirections.resize(m_ViewportWidth * m_ViewportHeight);
}

void Camera::ComputeRayDirections() noexcept {
    float verticalFovInRadians = Math::ToRadians(m_VerticalFovInDegrees);

    Math::Vector3f forward = m_Target - m_Position;
    float focalLength = Math::Length(forward);
    float viewportWorldHeight = 2.f * Math::Tan(verticalFovInRadians * 0.5f) * focalLength;
    float viewportWorldWidth = (viewportWorldHeight * m_ViewportWidth) / m_ViewportHeight;

    Math::Vector3f w = Math::Normalize(forward);
    Math::Vector3f u = Math::Normalize(Math::Cross(w, m_Up));
    Math::Vector3f v = Math::Normalize(Math::Cross(w, u));

    Math::Vector3f horizontal = u * viewportWorldWidth;
    Math::Vector3f vertical = v * viewportWorldHeight;
    Math::Vector3f leftUpper = forward - horizontal * 0.5f - vertical * 0.5f;

    for (int i = 0; i < m_ViewportHeight; ++i) {
        for (int j = 0; j < m_ViewportWidth; ++j) {
            float uScale = (float)(j + Utilities::RandomFloatInNegativeHalfToHalf()) / (m_ViewportWidth - 1);
            float vScale = (float)(i + Utilities::RandomFloatInNegativeHalfToHalf()) / (m_ViewportHeight - 1);
            m_RayDirections[m_ViewportWidth * i + j] = Math::Normalize(leftUpper + horizontal * uScale + vertical * vScale);
        }
    }
}
```

4.10 Camera.h

```
#ifndef _CAMERA_H
#define _CAMERA_H

#include "math/Math.h"

#include <vector>
#include <span>

class Camera {
public:
    inline Camera() noexcept = default;

    Camera(int viewportWidth, int viewportHeight, const Math::Vector3f &position = {0.f, 0.f, 0.f}, const Math::Vector3f &target = {0.f, 0.f, -1.f}, float verticalFovInDegrees = 20.f, const Math::Vector3f &up = {0.f, 1.f, 0.f}) noexcept;

    ~Camera() noexcept = default;

    void OnViewportResize(int viewportWidth, int viewportHeight) noexcept;

    void ComputeRayDirections() noexcept;

    constexpr std::span<const Math::Vector3f> GetRayDirections() const noexcept {
        return m_RayDirections;
    }

    constexpr Math::Vector3f GetPosition() const noexcept {
        return m_Position;
    }

    constexpr Math::Vector3f& Position() noexcept {
        return m_Position;
    }

    constexpr Math::Vector3f GetTarget() const noexcept {
        return m_Target;
    }

    constexpr Math::Vector3f& Target() noexcept {
        return m_Target;
    }

    constexpr Math::Vector3f GetUp() const noexcept {
        return m_Up;
    }

    constexpr Math::Vector3f& Up() noexcept {
        return m_Up;
    }

    constexpr float GetVerticalFovInDegrees() const noexcept {
        return m_VerticalFovInDegrees;
    }

    constexpr float& VerticalFovInDegrees() noexcept {
        return m_VerticalFovInDegrees;
    }

private:
    Math::Vector3f m_Position;
    Math::Vector3f m_Target;
    Math::Vector3f m_Up;
    float m_VerticalFovInDegrees;
    int m_ViewportWidth, m_ViewportHeight;

    std::vector<Math::Vector3f> m_RayDirections;
};

#endif
```

4.11 Entrypoint.cpp

```
#include "Application.h"

#include <iostream>

int main(int argc, char **argv) {
    int width = 1280, height = 720;
    if (argc > 1 && argc != 3) {
        std::cerr « "To specify initial window parametes use: [width] [height]\n";
        return -1;
    } else if (argc == 3) {
        width = atoi(argv[0]);
        height = atoi(argv[1]);
    }

    Application application(width, height);

    return application.Run();
}
```

4.12 HitPayload.h

```
#ifndef _HIT_PAYLOAD_H
#define _HIT_PAYLOAD_H

#include "math/Math.h"
#include "Material.h"

struct HitPayload {
    float t;
    Math::Vector3f normal;
    const Material *material;
};

#endif
```

4.13 Image.cpp

```
#include "Image.h"
#include <gl/gl.h>

Image::Image(int width, int height, const uint32_t *data) noexcept :
    m_Width(width), m_Height(height) {
    glGenTextures(1, &m_Descriptor);
    glBindTexture(GL_TEXTURE_2D, m_Descriptor);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_Width, m_Height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
(void*)data);

    glBindTexture(GL_TEXTURE_2D, 0);
}

Image::~Image() noexcept {
    glDeleteTextures(1, &m_Descriptor);
}

void Image::UpdateData(const uint32_t *data) noexcept {
    glBindTexture(GL_TEXTURE_2D, m_Descriptor);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, m_Width, m_Height, GL_RGBA, GL_UNSIGNED_BYTE, (void*)data);

    glBindTexture(GL_TEXTURE_2D, 0);
}
```

4.14 Image.h

```
#ifndef _IMAGE_H
#define _IMAGE_H

#include <vector>
#include <cstdint>

class Image {
public:
    Image() = delete;

    Image(int width, int height, const uint32_t *data) noexcept;

    ~Image() noexcept;

    void UpdateData(const uint32_t *data) noexcept;

    constexpr unsigned int GetDescriptor() const noexcept {
        return m_Descriptor;
    }
private:
    int m_Width, m_Height;
    unsigned int m_Descriptor;
};

#endif
```

4.15 Light.h

```
#ifndef _LIGHT_H
#define _LIGHT_H

#include "hittable/HittableObject.h"
#include "math/Math.h"
#include "Ray.h"
#include "HitPayload.h"

class Light {
public:
    constexpr Light(const HittableObjectPtr object, const Math::Vector3f &emission) noexcept :
        m_Object(object), m_Emission(emission) {}

    const HittableObject* GetObject() const noexcept {
        return m_Object;
    }

    Math::Vector3f Sample(const Ray &lightRay, const HitPayload &objectHitPayload, const HitPayload &lightHitPayload, float distance, float distanceSquared) const noexcept {
        constexpr float distanceEpsilon = 0.01f;
        if (Math::Abs(lightHitPayload.t - distance) > distanceEpsilon) {
            return Math::Vector3f(0.f);
        }

        float pdf = distanceSquared / (Math::Dot(lightHitPayload.normal, -lightRay.direction) * m_Object->GetArea());

        constexpr float pdfEpsilon = 0.01f;
        if (pdf <= pdfEpsilon) {
            return Math::Vector3f(0.f);
        }

        Math::Vector3f brdf = objectHitPayload.material->albedo * Math::Constants::InversePi<float> * Math::Dot(objectHitPayload.normal, lightRay.direction) * m_Emission;

        return brdf / pdf;
    }

private:
    const HittableObject* m_Object;
    Math::Vector3f m_Emission;
};

#endif
```

4.16 Material.h

```
#ifndef _MATERIAL_H
#define _MATERIAL_H

#include "math/Math.h"

struct Material {
    Math::Vector3f albedo;
    float metallic;
    float specular;
    float roughness;
    float emissionPower;

    int index;

    constexpr Math::Vector3f GetEmission() const noexcept {
        return albedo * emissionPower;
    }
};

#endif
```

4.17 Ray.h

```
#ifndef _RAY_H
#define _RAY_H

#include "math/Math.h"

struct Ray {
    Math::Vector3f origin, direction;
};

#endif
```

4.18 Renderer.cpp

```
#include "Renderer.h"
#include "Utilities.hpp"
#include "BSDF.h"

#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "../stb-master/stb_image_write.h"

#include <vector>
#include <thread>
#include <cstring>

Renderer::Renderer(int width, int height) noexcept :
    m_Width(width), m_Height(height) {
    m_ImageData = new uint32_t[m_Width * m_Height];
    m_Image = new Image(m_Width, m_Height, m_ImageData);
    m_AccumulationData = new Math::Vector4f[m_Width * m_Height];

    m_AvailableThreads = std::thread::hardware_concurrency();
    m_UsedThreads = 1;
    m_LinesPerThread = (m_Height + m_UsedThreads - 1) / m_UsedThreads;
}

Renderer::~Renderer() noexcept {
    if (m_Image != nullptr) {
        delete m_Image;
    }
    if (m_ImageData != nullptr) {
        delete[] m_ImageData;
    }
    if (m_AccumulationData != nullptr) {
        delete[] m_AccumulationData;
    }
}
```

```

void Renderer::SaveImage(const char *filename) const noexcept {
    stbi_write_png(filename, m_Width, m_Height, 4, m_ImageData, m_Width * 4);
}

void Renderer::OnResize(int width, int height) noexcept {
    if (m_Width == width && m_Height == height) {
        return;
    }

    m_Width = width;
    m_Height = height;

    if (m_ImageData != nullptr) {
        delete m_ImageData;
        m_ImageData = new uint32_t[m_Width * m_Height];
    }
    if (m_Image != nullptr) {
        delete m_Image;
        m_Image = new Image(m_Width, m_Height, m_ImageData);
    }
    if (m_AccumulationData != nullptr) {
        delete m_AccumulationData;
        m_AccumulationData = new Math::Vector4f[m_Width * m_Height];
    }
}

void Renderer::Render(const Camera &camera, std::span<const HittableObjectPtr> objects,
std::span<const Light> lightSources, std::span<const Material> materials) noexcept {
    m_Camera = &camera;
    m_Objects = objects;
    m_LightSources = lightSources;
    m_Materials = materials;

    if (!m_Accumulate) {
        m_FrameIndex = 1;
    }

    if (m_FrameIndex == 1) {
        memset(m_AccumulationData, 0, m_Width * m_Height * sizeof(Math::Vector4f));
    }

    float inverseFrameIndex = 1.f / m_FrameIndex;
    float inverseGamma = 1.f / m_Gamma;

    std::vector<std::thread> handles;
    handles.reserve(m_UsedThreads);

    for (int i = 0; i < m_Height; i += m_LinesPerThread) {
        handles.emplace_back([this, i, inverseFrameIndex, inverseGamma]() {
            int nextBlock = i + m_LinesPerThread;
            int limit = Math::Min(nextBlock, m_Height);
            for (int t = i; t < limit; ++t) {
                for (int j = 0; j < m_Width; ++j) {
                    m_AccumulationData[m_Width * t + j] += PixelProgram(t, j);

                    Math::Vector4f color = m_AccumulationData[m_Width * t + j];

                    color *= inverseFrameIndex;
                    color = Utilities::CorrectGamma(color, inverseGamma);
                    color = Math::Clamp(color, 0.f, 1.f);

                    m_ImageData[m_Width * t + j] = Utilities::ConvertColorToRGBA(color);
                }
            }
        });
    }

    for (auto &handle : handles) {
        handle.join();
    }

    m_Image->UpdateData(m_ImageData);

    if (m_Accumulate) {
        ++m_FrameIndex;
    }
}

void Renderer::Render(const Camera &camera, const AccelerationStructure &accelerationStructure,
std::span<const Light> lightSources, std::span<const Material> materials) noexcept {
    m_Camera = &camera;
    m_AccelerationStructure = &accelerationStructure;
    m_LightSources = lightSources;
    m_Materials = materials;

    if (!m_Accumulate) {
        m_FrameIndex = 1;
    }

    if (m_FrameIndex == 1) {
        memset(m_AccumulationData, 0, m_Width * m_Height * sizeof(Math::Vector4f));
    }

    float inverseFrameIndex = 1.f / m_FrameIndex;
    float inverseGamma = 1.f / m_Gamma;

    std::vector<std::thread> handles;
    handles.reserve(m_UsedThreads);

    for (int i = 0; i < m_Height; i += m_LinesPerThread) {
        handles.emplace_back([this, i, inverseFrameIndex, inverseGamma]() {
            int nextBlock = i + m_LinesPerThread;
            int limit = Math::Min(nextBlock, m_Height);
            for (int t = i; t < limit; ++t) {
                for (int j = 0; j < m_Width; ++j) {
                    m_AccumulationData[m_Width * t + j] += AcceleratedPixelProgram(t, j);

                    Math::Vector4f color = m_AccumulationData[m_Width * t + j];

                    color *= inverseFrameIndex;
                    color = Utilities::CorrectGamma(color, inverseGamma);
                    color = Math::Clamp(color, 0.f, 1.f);

                    m_ImageData[m_Width * t + j] = Utilities::ConvertColorToRGBA(color);
                }
            }
        });
    }

    for (auto &handle : handles) {
        handle.join();
    }

    m_Image->UpdateData(m_ImageData);

    if (m_Accumulate) {
        ++m_FrameIndex;
    }
}

Math::Vector4f Renderer::PixelProgram(int i, int j) const noexcept {
    Ray ray;
    ray.origin = m_Camera->GetPosition();
    ray.direction = m_Camera->GetRayDirections()[m_Width * i + j];

    Math::Vector3f light(0.f), throughput(1.f);
    for (int i = 0; i < m_RayDepth; ++i) {
        HitPayload payload = TraceRay(ray);

        if (payload.t < 0.f) {
            light += throughput * m_OnRayMiss(ray);
            break;
        }

        const Material *material = payload.material;
        auto emission = material->GetEmission();

        light += emission * throughput;

        if (material->emissionPower > 0.f) {
            break;
        }

        auto hitPoint = ray.origin + ray.direction * payload.t;
        for (auto lightSource : m_LightSources) {
            auto pointOnLight = lightSource.GetObject()->SampleUniform({Utilities::RandomFloatInZeroToOne(), Utilities::RandomFloatInZeroToOne()});

            auto toLight = pointOnLight - hitPoint;
            float distanceSquared = Math::Dot(toLight, toLight);
            float distance = Math::Sqrt(distanceSquared);

            Ray lightRay(hitPoint, toLight / distance);

            HitPayload lightHitPayload = TraceRay(lightRay);

            light += throughput * lightSource.Sample(lightRay, payload, lightHitPayload, distance, distanceSquared);
        }

        BSDF bsdf(material);
        auto direction = bsdf.Sample(ray, payload, throughput);

        ray.origin = hitPoint;
        ray.direction = direction;
    }

    return {light.r, light.g, light.b, 1.f};
}

Math::Vector4f Renderer::AcceleratedPixelProgram(int i, int j) const noexcept {
    Ray ray;
    ray.origin = m_Camera->GetPosition();
    ray.direction = m_Camera->GetRayDirections()[m_Width * i + j];

    Math::Vector3f light(0.f), throughput(1.f);
    for (int i = 0; i < m_RayDepth; ++i) {
        HitPayload payload = AcceleratedTraceRay(ray);

        if (payload.t < 0.f) {
            light += throughput * m_OnRayMiss(ray);
            break;
        }

        const Material *material = payload.material;
        Math::Vector3f emission = material->GetEmission();

        light += emission * throughput;

        if (material->emissionPower > 0.f) {
            break;
        }

        auto hitPoint = ray.origin + ray.direction * payload.t;
        for (auto lightSource : m_LightSources) {
            auto pointOnLight =
lightSource.GetObject()->SampleUniform({Utilities::RandomFloatInZeroToOne(),
Utilities::RandomFloatInZeroToOne()});

            auto toLight = pointOnLight - hitPoint;
            float distanceSquared = Math::Dot(toLight, toLight);
            float distance = Math::Sqrt(distanceSquared);

            Ray lightRay(hitPoint, toLight / distance);

            HitPayload lightHitPayload = TraceRay(lightRay);

            light += throughput * lightSource.Sample(lightRay, payload, lightHitPayload, distance,
distanceSquared);
        }

        BSDF bsdf(material);
        auto direction = bsdf.Sample(ray, payload, throughput);

        ray.origin += ray.direction * payload.t;
        ray.direction = direction;
    }

    return {light.r, light.g, light.b, 1.f};
}

HitPayload Renderer::TraceRay(const Ray &ray) const noexcept {
    HitPayload payload;
    payload.t = Math::Constants::Infinity<float>;
    payload.normal = Math::Vector3f(0.f);
    payload.material = nullptr;

    int objectCount = (int)m_Objects.size();
    for (int i = 0; i < objectCount; ++i) {
        m_Objects[i]->Hit(ray, 0.01f, Math::Min(payload.t, Math::Constants::Infinity<float>),
payload);
    }

    if (payload.material == nullptr) {
        return Miss(ray);
    }

    payload.normal = Math::Dot(ray.direction, payload.normal) > Math::Constants::Epsilon<float> ?
-payload.normal : payload.normal;

    return payload;
}

HitPayload Renderer::AcceleratedTraceRay(const Ray &ray) const noexcept {
    HitPayload payload;
    payload.t = Math::Constants::Infinity<float>;
    payload.normal = Math::Vector3f(0.f);
    payload.material = nullptr;

    m_AccelerationStructure->Hit(ray, 0.01f, Math::Constants::Infinity<float>, payload);

    if (payload.material == nullptr) {
        return Miss(ray);
    }

    payload.normal = Math::Dot(ray.direction, payload.normal) > Math::Constants::Epsilon<float> ? -payload.normal : payload.normal;

    return payload;
}

HitPayload Renderer::Miss(const Ray &ray) const noexcept {
    HitPayload payload;
    payload.t = -1.f;
    return payload;
}
```

4.19 Renderer.h

```
#ifndef _RENDERER_H
#define _RENDERER_H

#include "Image.h"
#include "Camera.h"
#include "Scene.h"
#include "HitPayload.h"
#include "Ray.h"
#include "math/Math.h"
#include "AccelerationStructure.h"
#include "Material.h"
#include "Light.h"

#include <functional>
#include <span>

class Renderer {
public:
    Renderer() = delete;

    Renderer(int width, int height) noexcept;

    ~Renderer() noexcept;

    using typ_t = HittableObject*;

    void Render(const Camera &camera, std::span<const HittableObjectPtr> objects, std::span<const Light> lightSources, std::span<const Material> materials) noexcept;

    void Render(const Camera &camera, const AccelerationStructure &accelerationStructure, std::span<const Light> lightSources, std::span<const Material> materials) noexcept;

    constexpr bool& Accumulate() noexcept {
        return m_Accumulate;
    }

    constexpr bool& Accelerate() noexcept {
        return m_Accelerate;
    }

    constexpr int GetFrameIndex() const noexcept {
        return m_FrameIndex;
    }

    constexpr Image* GetImage() const noexcept {
        return m_Image;
    }

    void SaveImage(const char *filename) const noexcept;

    void OnResize(int width, int height) noexcept;

    constexpr int GetAvailableThreadCount() const noexcept {
        return m_AvailableThreads;
    }

    constexpr int GetUsedThreadCount() const noexcept {
        return m_UsedThreads;
    }

    constexpr void SetUsedThreadCount(int usedThreads) noexcept {
        m_UsedThreads = Math::Clamp(usedThreads, 1, m_AvailableThreads);
        m_LinesPerThread = (m_Height + m_UsedThreads - 1) / m_UsedThreads;
    }

    constexpr int& UsedThreadCount() noexcept {
        return m_UsedThreads;
    }

    inline void OnRayMiss(std::function<Math::Vector3f(const Ray&)> onRayMiss) noexcept {
        m_OnRayMiss = onRayMiss;
    }

    constexpr int& RayDepth() noexcept {
        return m_RayDepth;
    }

    constexpr float& Gamma() noexcept {
        return m_Gamma;
    }

private:
    Math::Vector4f PixelProgram(int u, int j) const noexcept;

    Math::Vector4f AcceleratedPixelProgram(int i, int j) const noexcept;

    HitPayload TraceRay(const Ray &ray) const noexcept;

    HitPayload AcceleratedTraceRay(const Ray &ray) const noexcept;

    HitPayload Miss(const Ray &ray) const noexcept;

private:
    int m_Width, m_Height;
    Image *m_Image = nullptr;
    uint32_t *m_ImageData = nullptr;

    std::function<Math::Vector3f(const Ray&)> m_OnRayMiss = [](const Ray&){ return Math::Vector3f(0.f, 0.f, 0.f); };

    int m_AvailableThreads;
    int m_UsedThreads;
    int m_LinesPerThread;

    int m_RayDepth = 5;

    const Camera *m_Camera = nullptr;
    std::span<const HittableObjectPtr> m_Objects;
    std::span<const Light> m_LightSources;
    const AccelerationStructure *m_AccelerationStructure = nullptr;
    std::span<const Material> m_Materials;

    bool m_Accumulate = false;
    Math::Vector4f *m_AccumulationData = nullptr;
    int m_FrameIndex = 1;

    bool m_Accelerate = false;

    float m_Gamma = 2.f;
};

#endif
```

4.20 Scene.h

```
#ifndef _SCENE_H
#define _SCENE_H

#include "hittable/Sphere.h"
#include "hittable/Triangle.h"
#include "hittable/Box.h"
#include "hittable/Model.h"
#include "Material.h"
#include "Camera.h"

#include <vector>
#include <fstream>
#include <array>
#include <exception>
#include <optional>

struct Scene {
    std::vector<Shapes::Sphere> spheres;
    std::vector<Shapes::Triangle> triangles;
    std::vector<Shapes::Box> boxes;
    std::vector<Model*> models;
    std::vector<Material> materials;
    Camera camera;

    std::optional<std::string> Serialize(std::ostream &os) const noexcept {
        os.exceptions(std::ios::badbit | std::ios::failbit);

        try {
            TrySerialize(os);
        } catch (std::exception &e) {
            return e.what();
        }

        return {};
    }

    std::optional<std::string> Deserialize(std::istream &is) noexcept {
        is.exceptions(std::ios::eofbit | std::ios::badbit | std::ios::failbit);

        try {
            TryDeserialize(is);
        } catch (std::exception &e) {
            return e.what();
        }

        return {};
    }

private:
    void TrySerialize(std::ostream &os) const {
        int materialCount = static_cast<int>(materials.size());
        int sphereCount = static_cast<int>(spheres.size());
        int triangleCount = static_cast<int>(triangles.size());
        int boxCount = static_cast<int>(boxes.size());
        int modelCount = static_cast<int>(models.size());

        os.write(reinterpret_cast<const char*>(&materialCount), sizeof(materialCount));
        os.write(reinterpret_cast<const char*>(&sphereCount), sizeof(sphereCount));
        os.write(reinterpret_cast<const char*>(&triangleCount), sizeof(triangleCount));
        os.write(reinterpret_cast<const char*>(&boxCount), sizeof(boxCount));
        os.write(reinterpret_cast<const char*>(&modelCount), sizeof(modelCount));

        for (const auto &material : materials) {
            os.write(reinterpret_cast<const char*>(&material), sizeof(material));
        }

        for (const auto &sphere : spheres) {
            os.write(reinterpret_cast<const char*>(&sphere.center), sizeof(sphere.center));
            os.write(reinterpret_cast<const char*>(&sphere.radius), sizeof(sphere.radius));
            os.write(reinterpret_cast<const char*>(&sphere.material->index), sizeof(sphere.material->index));
        }

        for (const auto &triangle : triangles) {
            os.write(reinterpret_cast<const char*>(&triangle.vertices), sizeof(triangle.vertices));
            os.write(reinterpret_cast<const char*>(&triangle.normal), sizeof(triangle.normal));
            os.write(reinterpret_cast<const char*>(&triangle.material->index), sizeof(triangle.material->index));
        }

        for (const auto &box : boxes) {
            os.write(reinterpret_cast<const char*>(&box.min), sizeof(box.min));
            os.write(reinterpret_cast<const char*>(&box.max), sizeof(box.max));
            os.write(reinterpret_cast<const char*>(&box.material->index), sizeof(box.material->index));
        }

        for (const auto model : models) {
            auto pathToFile = model->GetPathToFile().string();
            int pathToFileLength = static_cast<int>(pathToFile.length());

            auto materialDirectory = model->GetMaterialDirectory().string();
            int materialDirectoryLength = static_cast<int>(materialDirectory.length());

            os.write(reinterpret_cast<const char*>(&pathToFileLength), sizeof(pathToFileLength));
            os.write(pathToFile.data(), pathToFileLength);

            os.write(reinterpret_cast<const char*>(&materialDirectoryLength), sizeof(materialDirectoryLength));
            os.write(materialDirectory.data(), materialDirectoryLength);
        }

        // todo! loop for models here

        auto position = camera.GetPosition();
        auto target = camera.GetTarget();
        auto verticalFovInDegrees = camera.GetVerticalFovInDegrees();
        auto up = camera.GetUp();

        os.write(reinterpret_cast<const char*>(&position), sizeof(position));
        os.write(reinterpret_cast<const char*>(&target), sizeof(target));
        os.write(reinterpret_cast<const char*>(&verticalFovInDegrees), sizeof(verticalFovInDegrees));
        os.write(reinterpret_cast<const char*>(&up), sizeof(up));
    }

    void TryDeserialize(std::istream &is) {
        int materialCount;
        int sphereCount;
        int triangleCount;
        int boxCount;
        int modelCount;

        is.read(reinterpret_cast<char*>(&materialCount), sizeof(materialCount));
        is.read(reinterpret_cast<char*>(&sphereCount), sizeof(sphereCount));
        is.read(reinterpret_cast<char*>(&triangleCount), sizeof(triangleCount));
        is.read(reinterpret_cast<char*>(&boxCount), sizeof(boxCount));
        is.read(reinterpret_cast<char*>(&modelCount), sizeof(modelCount));

        materials.clear();
        materials.resize(materialCount);
        for (auto &material : materials) {
            is.read(reinterpret_cast<char*>(&material), sizeof(material));
        }
        materials.shrink_to_fit();

        spheres.clear();
        spheres.reserve(sphereCount);
        while (sphereCount--) {
            Math::Vector3f center;
            float radius;
            int materialIndex;
            is.read(reinterpret_cast<char*>(&center), sizeof(center));
            is.read(reinterpret_cast<char*>(&radius), sizeof(radius));
            is.read(reinterpret_cast<char*>(&materialIndex), sizeof(materialIndex));

            spheres.emplace_back(center, radius, &materials[materialIndex]);
        }
        spheres.shrink_to_fit();

        triangles.clear();
        triangles.reserve(triangleCount);
        while (triangleCount--) {
            Math::Vector3f vertices[3];
            Math::Vector3f normal;
            int materialIndex;
            is.read(reinterpret_cast<char*>(&vertices), sizeof(vertices));
            is.read(reinterpret_cast<char*>(&normal), sizeof(normal));
            is.read(reinterpret_cast<char*>(&materialIndex), sizeof(materialIndex));

            triangles.emplace_back(std::array<Math::Vector3f, 3>{vertices[0], vertices[1], vertices[2]}, normal, &materials[materialIndex]);
        }
        triangles.shrink_to_fit();

        boxes.clear();
        boxes.reserve(boxCount);
        while (boxCount--) {
            Math::Vector3f min, max;
            int materialIndex;
            is.read(reinterpret_cast<char*>(&min), sizeof(min));
            is.read(reinterpret_cast<char*>(&max), sizeof(max));
            is.read(reinterpret_cast<char*>(&materialIndex), sizeof(materialIndex));

            boxes.emplace_back(min, max, &materials[materialIndex]);
        }
        boxes.shrink_to_fit();


        for (const auto model : models) {
            delete model;
        }

        models.clear();
        models.reserve(modelCount);
        while (modelCount--) {
            int pathToFileLength;
            is.read(reinterpret_cast<char*>(&pathToFileLength), sizeof(pathToFileLength));

            std::vector<char> buffer(pathToFileLength);
            is.read(buffer.data(), pathToFileLength);

            std::string pathToFile(buffer.data(), pathToFileLength);

            int materialDirectoryLength;
            is.read(reinterpret_cast<char*>(&materialDirectoryLength),
sizeof(materialDirectoryLength));

            buffer.resize(materialDirectoryLength);
            is.read(buffer.data(), materialDirectoryLength);

            std::string materialDirectory(buffer.data(), materialDirectoryLength);

            auto result = Model::LoadOBJ(pathToFile, materialDirectory);
            if (result.IsFailure()) {
                continue;
            }

            models.push_back(result.model);
        }
        models.shrink_to_fit();

        is.read(reinterpret_cast<char*>(&camera.Position()), sizeof(camera.Position()));
        is.read(reinterpret_cast<char*>(&camera.Target()), sizeof(camera.Target()));
        is.read(reinterpret_cast<char*>(&camera.VerticalFovInDegrees()), sizeof(camera.VerticalFovInDegrees()));
        is.read(reinterpret_cast<char*>(&camera.Up()), sizeof(camera.Up()));
    }
};

#endif
```

4.21 Timer.h

```
#ifndef _TIMER_H
#define _TIMER_H

#include <chrono>
#include <functional>

class Timer {
public:
    constexpr Timer() noexcept = delete;
    constexpr Timer(const Timer&) = delete;
    constexpr Timer(Timer&&) = delete;

    inline static double MeasureInMillis(std::function<void()> f) {
        auto t1 = std::chrono::high_resolution_clock::now();
        f();
        auto t2 = std::chrono::high_resolution_clock::now();
        static_cast<std::chrono::duration<float, std::milli>>(t2 - t1).count();
        return std::chrono::duration_cast<std::chrono::duration<float, std::milli»(t2 - t1).count();
    }
};

#endif
```

4.22 Utilities.hpp

```
#ifndef _UTILITIES_HPP
#define _UTILITIES_HPP

#include <chrono>
#include <random>

#include "math/Math.h"

namespace Utilities {
    inline static uint32_t s_RandomEngineState = std::chrono::high_resolution_clock::now().time_since_epoch().count();
    thread_local inline static std::mt19937_64 s_RandomNumberGenerator(std::chrono::high_resolution_clock::now().time_since_epoch().count());
    inline static std::uniform_real_distribution<> s_ZeroToOne(0.f, 1.f);

    inline uint32_t RandomUint() {
        uint32_t state = s_RandomEngineState;
        s_RandomEngineState = state * 747796405u + 2891336453u;
        uint32_t word = ((state » ((state » 28u) + 4u)) ^ state) * 277803737u;
        return (word » 22u) ^ word;
    }

    inline float RandomFloatInZeroToOne() noexcept {
        return s_ZeroToOne(s_RandomNumberGenerator);
    }

    inline float RandomFloatInNegativeHalfToHalf() noexcept {
        return RandomFloatInZeroToOne() - 0.5f;
    }

    inline float RandomFloatInNegativeToOne() noexcept {
        return RandomFloatInZeroToOne() * 2.f - 1.f;
    }

    inline int RandomIntInRange(int min, int max) noexcept {
        return std::uniform_int_distribution<>(min, max)(s_RandomNumberGenerator);
    }

    inline Math::Vector3f RandomInUnitSphere() noexcept {
        while (true) {
            Math::Vector3f result(RandomFloatInNegativeToOne(), RandomFloatInNegativeToOne(),
RandomFloatInNegativeToOne());
            if (Math::Dot(result, result) < 1.f) {
                return result;
            }
        }
    }

    inline Math::Vector3f RandomUnitVector() noexcept {
        return Math::Normalize(RandomInUnitSphere());
    }

    inline Math::Vector3f RandomInHemisphere(const Math::Vector3f &normal) {
        Math::Vector3f randomUnitVector = RandomUnitVector();
        if (Math::Dot(randomUnitVector, normal) < 0.f) {
            return -randomUnitVector;
        }

        return randomUnitVector;
    }

    constexpr float InverseSqrtFast(float x) {
        constexpr float threeHalfs = 1.5f;

        float halfX = x * 0.5f;
        uint32_t i = *(uint32_t*)&x;
        i = 0x5f3759df - (i >> 1);

        x = *(float*)&i;
        x = x * (threeHalfs - halfX * x * x);

        return x;
    }

    inline Math::Vector3f RandomUnitVectorFast() noexcept {
        Math::Vector3f v = RandomInUnitSphere();
        return v * InverseSqrtFast(Math::Dot(v, v));
    }

    inline Math::Vector3f RandomInHemisphereFast(const Math::Vector3f &normal) {
        Math::Vector3f randomUnitVector = RandomUnitVectorFast();
        if (Math::Dot(randomUnitVector, normal) < 0.f) {
            return -randomUnitVector;
        }

        return randomUnitVector;
    }

    inline Math::Vector3f RandomCosineDirection() {
        float r1 = RandomFloatInZeroToOne();
        float r2 = RandomFloatInZeroToOne();

        constexpr float twoPi = 2.f * std::numbers::pi;

        float phi = twoPi * r1;
        float x = cos(phi) * sqrt(r2);
        float y = sin(phi) * sqrt(r2);
        float z = sqrt(1.f - r2);

        return {x, y, z};
    }

    constexpr uint32_t AsUint(float x) {
        return *(uint32_t*)&x;
    }

    constexpr float AsFloat(uint32_t x) {
        return *(float*)&x;
    }

    constexpr float PowFast(float x, float exp) {
        constexpr float oneAsUint = 0x3f800000u;
        return AsFloat(int(exp * (AsUint(x) - oneAsUint)) + oneAsUint);
    }

    constexpr bool AlmostZero(const Math::Vector3f &v) {
        constexpr float epsilon = std::numeric_limits<float>::epsilon();
        return Math::Abs(v.x) < epsilon && Math::Abs(v.y) < epsilon && Math::Abs(v.z) < epsilon;
    }

    constexpr uint32_t ConvertColorToRGBA(const Math::Vector4f &color) noexcept {
        uint8_t r = (uint8_t)(color.r * 255.0f);
        uint8_t g = (uint8_t)(color.g * 255.0f);
        uint8_t b = (uint8_t)(color.b * 255.0f);
        uint8_t a = (uint8_t)(color.a * 255.0f);

        return (a « 24) | (b « 16) | (g « 8) | r;
    }

    constexpr Math::Vector4f CorrectGamma(const Math::Vector4f &color, float inverseGamma) {
        return {
            Math::Pow(color.r, inverseGamma),
            Math::Pow(color.g, inverseGamma),
            Math::Pow(color.b, inverseGamma),
            color.a
        };
    }

    constexpr Math::Vector4f CorrectGammaFast(const Math::Vector4f &color, float inverseGamma) {
        return {
            PowFast(color.r, inverseGamma),
            PowFast(color.g, inverseGamma),
            PowFast(color.b, inverseGamma),
            color.a
        };
    }
}

#endif
```

Index