Prof. Dr. Philipp Slusallek
Dr. Pascal Grittmann
Teaching assistant: Lukas Auer

UNIVERSITÄT DES SAARLANDES

Due: 01 November 2026, 23:59

Computer Graphics
Practical Assignment 1

Submission instructions

Your solution must be pushed to your GitLab repository. By default, we will grade the last commit on the `main` branch before the due date.

Remember: While assignments are ungraded, passing all tests on the gitlab runner by the due date is strictly required for exam admission. Beyond that, do not forget to prepare the colloquium questions for your colloquium.

Asking questions

General questions regarding the assignment should be asked on Discourse (`Forum` tab in CMS)

1.1 Perspective Camera Model

The basic scaffolding for a perspective camera is provided in `src/cameras/perspective.cpp`. Your task is to implement the missing pieces of code in the constructor and in the ray generation (`sample`). You can (and should) add new member variables to precompute expensive quantities (such as trigonometric functions) in the constructor – `sample` will be called millions of times.

Initially, the constructor only queries two camera properties from the `.xml` scene description: The field of view (`fov`) and the axis on which this angle is defined (horizontal, `"x"` , or vertical, `"y"`). The image resolution is available in the base class `Camera`.

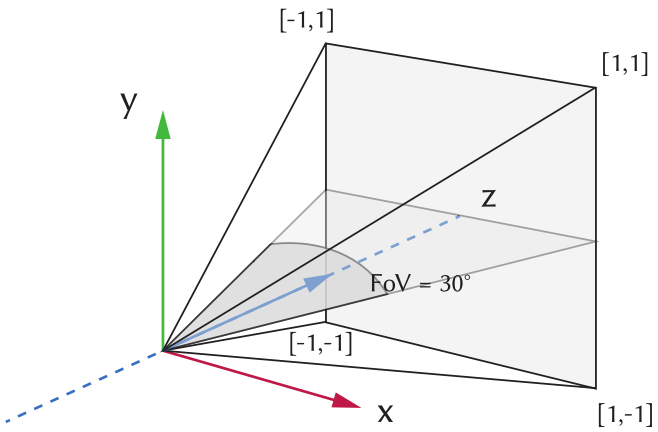

Figure 1: *The left-handed local camera coordinate system in which rays are defined before being transformed to world coordinates. Here, the field of view is 30 degrees and defined with respect to the x axis.*

The `sample` function takes as input normalized image coordinates in $[-1, +1]^2$ and should output transformed camera rays in world coordinates (the coordinate system where your scene lives). This is done in two steps:

a) Generate a ray in the camera coordinate system (see Figure 1). In camera space, the camera position is always the origin $(0, 0, 0)$ and the view direction is the positive z axis.

b) Apply the camera-to-world transformation `m_transform` specified in the scene description (`m_transform->apply(ray);`).

The direction returned by `sample` must be *normalized*.

Testing your implementation The following tests should all pass when running the `run_tests.py` script:

- `practical_1/perspective_20deg`
- `practical_1/perspective_90deg`
- `practical_1/perspective_90deg_translated`
- `practical_1/perspective_120deg_rotated`

The first two tests make sure your implementation is working when no transformation is applied to the camera. The last two should pass when you properly apply the transformation to the camera.

1.2 Basic Primitives and Normal Integrator

Now that you have a working camera model, it is time to add a few primitives to your renderer and display them.

1.2.1 Implementing a basic AOV integrator

When it comes to debugging your code, it is common practice to use an *arbitrary output variable* (AOV) integrator to visualize different scene properties. This integrator takes a single string parameter `variable` that decides which quantity should be visualized.

If you ran the tests in the previous tasks, you probably observed that many of the tests fail with the message `<integrator /> with unknown type="aov"`, which indicates that there is currently no integrator of type `aov`. To fix this, add an `aov.cpp` file in `src/integrators/` with the following minimal content:

```
#include <lightwave.hpp>

namespace lightwave {
class AovIntegrator : public SamplingIntegrator {
public:
    AovIntegrator(const Properties &properties) : SamplingIntegrator(properties) {}

    Color Li(const Ray &ray, Sampler &rng) override {
        NOT_IMPLEMENTED
    }

    std::string toString() const override {
        return tfm::format(
            "AovIntegrator[\n"
            "  sampler = %s,\n"
            "  image = %s,\n"
            "]",
            indent(m_sampler),
            indent(m_image));
      }
};
}// namespace lightwave
REGISTER_INTEGRATOR(AovIntegrator, "aov")
```

If you run the tests again, the previous error should be gone – unfortunately just to raise a `NOT_IMPLEMENTED` error, which you should fix in the next task.

The Normal Integrator The first mode you will implement for your AOV integrator visualizes normal vectors and should be activated if `variable` is set to `normals`. For a meaningful visualization, normals must be

remapped from $[-1, 1]^3$ to the interval $[0, 1]^3$, ensuring that normals with negative components can be displayed correctly. This mapping should be linear, i.e.,

$$n' = \frac{n+1}{2}$$

In the end, your implementation should perform the following steps:

- Intersect the ray against the scene to obtain intersection information.
- If an intersection occurs, record the shading normal at that point; otherwise use a zero vector.
- Remap the resulting normal as described above and return the result.

If your AOV integrator is implemented correctly, the following test should pass:

- `practical_1/aov_rectangle`

1.2.2 Ray-Sphere Intersection

Your next task is to fill in the provided skeleton in `src/shapes/sphere.cpp`. The `sphere` shape should be positioned at the origin $(0, 0, 0)$ and have radius 1. Matrix transformations (via instancing) are used to achieve spheres of other positions / sizes. To implement the intersect function, you will need to solve the following equations for t (this is exercise 1.1 on the theoretical sheet):

- Ray equation: $R(t) = o + t \cdot d$
- Sphere equation: $\|x - c\| = r$ (c is center, r radius)

Remember that `intersect` should return the *closest* hit and avoid self-intersection (as discussed in the lecture). The framework provides `ray.tMin` and `ray.tMax` to help with this. `ray.tMin` defines the minimum valid distance along the ray, preventing self-intersections, while `ray.tMax` stores the distance to the closest hit found so far. Therefore, when testing an intersection with the shape, you only need to consider hits whose distance lies within the interval [`tMin` , `tMax`]. If you find such a hit, update the intersection accordingly, including the `SurfaceEvent` fields, which are:

- `position` : the position of the found intersection in local cartesian coordinates.
- `uv` : a 2D parametrization of the hit point in $[0, 1]^2$ (a longitude/latitude mapping), computed as:

```
surf.uv = Vector2(
            (std::atan2(-surf.position.x(), -surf.position.z()) + Pi) * Inv2Pi,
            safe_acos(-surf.position.y()) * InvPi
          );
```

- `geometryNormal` : the outward-pointing unit surface normal.
- `shadingNormal` : the normal of the shading frame; on a sphere, this is identical to the geometry normal.
- `tangent` : any unit vector orthogonal to the normal (see Figure 2 for examples).
- `pdf` : the probability density of sampling this point (only needed later). For now, simply set it to 0.

`getBoundingBox` (computes the AABB) and `getCentroid` (computes the center of mass) have to be implemented as well, but `sampleArea` can remain `NOT_IMPLEMENTED` for now. If you're unsure about what some of the functions should return or how to implement them, you can refer to the implementation of `src/shapes/rectangle.cpp` for guidance.

If everything was implemented properly, the following tests should now pass:

- `practical_1/sphere_outside`
- `practical_1/sphere_inside`
- `practical_1/sphere_epsilon`

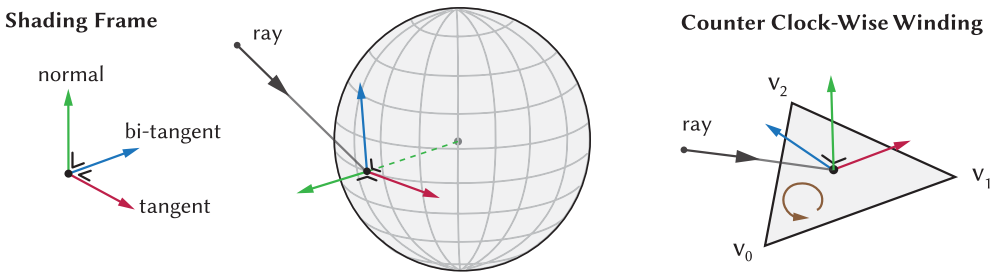

Figure 2: *Examples of valid shading frames in Lightwave for intersecting spheres and triangle meshes.*

1.2.3 Ray-Mesh Intersection

Spheres are nice, but complex objects are best represented using triangles. We already provide you with all the mechanisms required to load a triangle mesh from a `.ply` object file on disk. Loading of the PLY file is done in the constructor of a triangle mesh in `src/shapes/mesh.cpp`. Once loaded, the triangle data are stored in two vectors: `m_vertices`, which stores the 3D locations, normal and texture coordinates of every vertex in the mesh, and `m_triangles`, which stores the index to the three vertices in `m_vertices` that define the triangle.

Your task is again to implement the `intersect`, `getBoundingBox` and `getCentroid` functions in `src/shapes/mesh.cpp`, however, note that the function's signatures have changed a little and take an additional `primitiveIndex` parameter, which indicates the particular triangle to intersect. For the ray-triangle intersection itself, you should implement the Möller-Trumbore algorithm. You will also have to implement normal interpolation (as discussed in the lecture), which is set on or off using the `m_smoothNormals` property. To do so, you will need to interpolate normals based on the intersection's barycentric coordinates. Hint: you probably want to use the provided `Vertex::interpolate` function defined in `include/lightwave/math.hpp`.

Once everything is implemented, the following tests should now pass:

- `practical_1/mesh_bunny`
- `practical_1/mesh_flat`
- `practical_1/mesh_smooth`
- `practical_1/mesh_inside`

1.3 Instance Transform

Now that we have some basic shapes, we can implement instancing to (1) apply arbitrary transformations to basic shapes and (2) place multiple copies into the scene.

The framework already contains a partial implementation in `src/core/instance.cpp`, but `Instance::transformFrame` and `Instance::intersect` remain to be implemented. `intersect` must apply the inverse of `m_transform` to transform the given ray into object space, while `transformFrame` is supposed to transform the computed intersection from object space to world space.

Keep in mind that *normals* need special handling.

When finished, the following tests should pass:

- `practical_1/sphere_mirrored`
- `practical_1/sphere_scaled`
- `practical_1/sphere_sheared`
- `practical_1/sphere_snowman`
- `practical_1/mesh_bunny_flat`

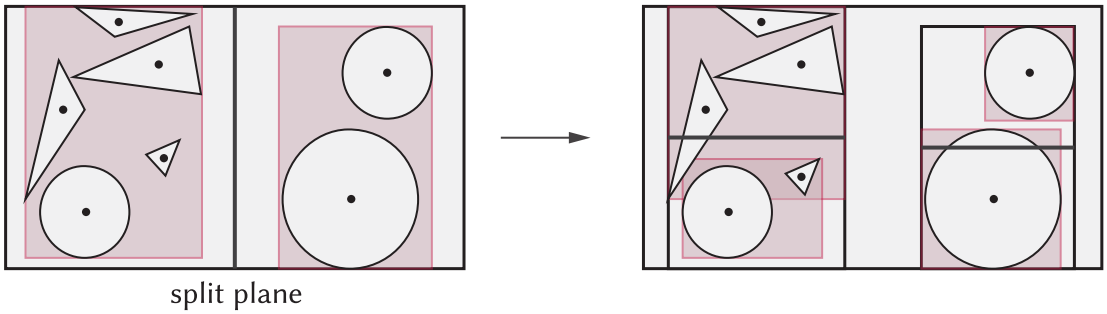

Figure 3: *Two steps of applying a "split-in-the-middle" heuristic for the BVH construction. Further details on this method can be found in the lecture slides.*

1.4 SAH Bounding Volume Hierarchy

To avoid tracing rays against every primitive in a scene, Lightwave uses a bounding volume hierarchy (BVH) as an acceleration data structure to reduce the number of ray intersection tests needed to intersect with the scene's geometry. You can find the implementation in `src/shapes/accel.hpp`. Currently, this implementation uses the naive *"split-in-the-middle" heuristic*, which is visualized in Figure 3. In this task you will replace this approach with the superior *surface area heuristic (SAH)* in combination with *binning* to accelerate intersection tests even further.

1.4.1 Extending the AOV Integrator

To debug the changes we make to the BVH construction in the following task, we first need a way to visualize such a BVH. To achieve this, the AOV integrator from Task 1.2.1 can be extended to visualize the BVH structure whenever the `variable` parameter is set to `bvh`.

The BVH Integrator To visualize the structure of a BVH, an intersection keeps track of some related statistics in the `stats` variable. The BVH integrator should use two of its components, `bvhCounter` and `primCounter`, to display a heat map representing BVH intersections. Since these values can become quite large, you need to extend your integrator by a `scale` parameter (default value: `1`), that will be used to keep colors within a reasonable range. In the end, your implementation should perform the following:

- Intersect the ray against the scene to obtain intersection information.
- Create a color with `bvhCounter` as the first and `primCounter` as the second component.
- Divide this color by `scale` and return the result.

1.4.2 SAH Binning

To add SAH binning to your renderer, you have to implement the `binning` function in `src/shapes/accel.hpp`. You can take a look at the exceptionally detailed blog post from Jacco Bikker to take inspiration for your implementation.

To test your implementation, you have to explicitly enable binning by setting `UseSAH` to `true`. If everything is implemented correctly, you should pass the following two tests *with a binning size of 16*:

- `practical_1/bvh_simple`
- `practical_1/bvh_complex`

1.5 Rename your Renderer!

You probably have noticed that the compiled binary *"unnamed"* is a very unimaginative name for a renderer. It is time to make your renderer your own and rename it however you like! We let you figure out how to do this (searching for `unnamed` in the codebase is probably a good start).