This handout was compiled on 2025-01-24 14:53:22+02:00.

CS-E5520 Advanced Computer Graphics, Spring 2025

Lehtinen / Härkönen, Timonen

Assignment 1: Accelerated Ray Tracing

Due February 16, 2025 at 23:59

In this assignment, you are provided with a basic implementation of a naive raytracer. It is capable of rendering images, but is prohibitively slow because it does not use an acceleration structure. Your task is to speed up the raytracer with a bounding volume hierarchy (BVH) and multithreading. In addition, you will add texturing to the default shading scheme to make it more interesting, and use your raytracer to implement a simple but effective shading trick called ambient occlusion (AO). It resembles full global illumination to an extent, both in appearance and methodology.

Requirements (maximum 10p) *on top of which you can do extra credit*

1. **BVH construction and traversal (5p)**
2. **BVH saving and loading (1p)**
3. **Simple texturing (1p)**
4. **Ambient occlusion (2p)**
5. **Simple multithreading (1p)**

1 Getting Started

1.1 Environment and tools

Those who have previously taken CS-C3100 Computer Graphics will be right at home; there are no substantial technology changes for this course; apart from the framework/math library code which we unfortunately did not have time to update for this semester. The assignment format is also the same: each assignment comes with a handout like this one, and some starter code where you fill in the missing functionality and turn your solutions in for grading.

We use Visual Studio 2022 to prepare and grade the assignments. You can either install the free Community edition on your own machine, or work in a classroom or the Aalto VDI machines "Win 3D" instances (link). Keep in mind that VDI compresses the video stream quite a bit so image quality might be less than ideal, especially for things that move. You are also free to use any other environment for development, but you will need to submit a working Visual Studio 2022 solution. In any case, be sure to test your solutions on the VDI before submitting; this is what we use for grading, and we expect your solution to work there.

Assignment solution files (`.sln`) contain two projects, `base` and `framework`. The `base` project contains the files specific to the current assignment; all files you need to modify are there. `framework` contains supporting code that stays the same between assignments.

For the first assignment, there is also a third project called `plotter`. It implements a small utility program that plots timing measurements in order to help evaluate performance of your ray tracing code; you're not required to run it, but it might be useful for tracking your optimization progress. The project is included so you can modify the program if you wish.

There are two build configurations in the Visual Studio solution you can choose from: Debug and Release. (To be precise, there are more: ReleaseAssert is release with asserts enabled, and there are two Solution configurations; they're artifacts of our system, you can just ignore them.) If your program feels slow, **always try running it in Release**

before assuming there's something wrong with your code; Debug builds can run orders of magnitude slower than Release builds. You'll probably find need for both Release and Debug builds, so don't stick to only one of them. Debug mode is really useful for catching haunting problems early on.

To handle some graphics and UI tasks, we use a utility framework from Nvidia Research which sits in the C++ namespace `FW`. `FW` code isn't documented and can be cryptic, but you can ignore most of it. The only part of `FW` you will need constantly is the math library (replacing Eigen from the intro class) found in `Math.hpp`, such as `FW::Vec3f` vectors and `FW::Mat4f` matrices. There's a separate handout in MyCourses on the basics of using the math library.

For graphics acceleration we use OpenGL. All assignment code should work on computers which support OpenGL 3.3 or better; if your GPU fills this requirement but assignment code still fails to work, bring it up on the course forum or Slack and we'll look into it.

We do not judge your code based on style or code quality as long as it works and implements the requirements. For your own sake, do try to keep things sane and consistent. Google naming style is one good option.

1.2 Sample solutions

We provide a sample executable for each assignment that you can use to see how a correct implementation should behave (`reference_assignment1.exe`). The viewer comes preloaded with three sample scenes, which can be loaded by pressing 1, 2 or 3 on the keyboard. You can fly around in the scene using the mouse, arrow keys and WASD keys. Use the mouse wheel to adjust the flight speed. You can save the current camera position and other settings to disk by pressing Alt and a number key; the saved state can later be loaded by pressing the corresponding number key. Pressing "Print Screen" hides the UI and saves a screenshot into a file.

There are several included scene files of various detail and possible optional features. Before implementing the acceleration structures you should only probably render the Cornell or Indirect test scenes which consist of a moderately low number of triangles. The conference scene is a good one to measure performance since it has a lot of details and narrow views; note that it's still a relatively simple scene by today's standards. To test texturing you should use sponza, since the previously mentioned scenes have no textures. If you want to do some of the texturing extras such as normal mapping, you should use the Crytek variant of the sponza model since it has all the required data readily available.

When you click the "Trace Rays" button (or press Enter), the program renders the model from the current viewpoint using the ray tracer. You can toggle between the last raytraced image and the real time viewer with the "Show Ray Tracer result" button or Space. The raytracer defaults to simple headlight shading (F1) that does not produce particularly pretty pictures. For a more interesting look, select "Ambient Occlusion shading" (F2) and render again. You can adjust the parameters of the AO effect using the "AO rays" and "AO ray length" sliders. Note that the size of the preview window determines the rendering resolution; shrinking it allows you to experiment more rapidly.

The example always uses the Surface Area Heuristic to optimize its trees when invoked from the GUI. This leads to a slow BVH building process and high tracing performance.

1.3 Base code

Next, you should compile the provided source code using Visual Studio and run it (preferably from the VS environment, so that the relative data paths will be correctly set up). Remember also to switch to the Release configuration unless you need the debug features. The solution file doesn't encode the startup project so it can be the wrong project; make sure that it is `Assignment1` – you can right click it and choose 'Set as StartUp Project' if it is not already. The base code has some of the functionality of the sample solution, but ambient occlusion rendering produces a blank image, and the ray tracer is very slow and texturing and more sophisticated rendering methods are not available. You should probably test it only in the very simple Cornell box scene at first; the Cornell box is also the most reasonable choice for finding errors in the debug mode.

Familiarize yourself with the code of the base solution. Majority of the code in the project manages the real-time viewer, mesh loading and other infrastructure; in addition, we provide a fast ray-triangle intersection routine. You do not need to understand this code in detail (nor modify it), unless you e.g. want to add new buttons for special features you have implemented. The relevant source files are `RayTracer.*` and `Renderer.*`. In particular, the header files give a concise overview of the different components in the renderer.

1.4 Batch rendering mode

In order to easily compare your results with the reference solution and see how fast your renderer or a particular method is, you can run a set of predetermined rendering tasks using the batch files in `timing_sets`. There are some simple examples and a readme file that explains the details of creating new measurement sets.

2 Detailed instructions

You can work on R1, R3 and R4 in any order, although the textured sponza scene and ambient occlusion are rather slow before you have a BVH. We recommend you enable multithreading (R5) before doing anything else, since it speeds up your testing and lets you see concurrency bugs in your code sooner. If you get weird bugs, remember to check whether they are concurrency issues by temporarily disabling multithreading!

R1 BVH construction and traversal (5p)

The overall ideas of the BVH construction and ray traversal are detailed in the lecture slides.

The BVH construction should be implemented in `RayTracer::constructHierarchy`. The function receives a reference to a vector containing all triangles in the scene. You should modify the BVH member variable stored by `RayTracer` to initialize the pointer to the root node of the BVH as well as any other modifications that are needed to build the hierarchy.

The starter code includes a simple BVH node class in BvhNode.hpp and BvhNode.cpp. You can use it as given, or you can modify it to fit your needs. If you do modify it, you won't be able to (easily) load a BVH generated by the example solution, and you'll have to implement BVH saving and loading yourself.

The declaration of BvhNode from BvhNode.hpp is shown below:

```
struct BvhNode
{
   AABB box;                      // Axis-aligned bounding box
   int startPrim, endPrim;        // Indices in the global list
   std::unique_ptr<BvhNode> left;
   std::unique_ptr<BvhNode> right;
}
```

`startPrim` and `endPrim` define the range of triangles contained by the node, while `left` and `right` are pointers to the node's children, if they exist. If the node is a leaf, the child pointers will be null.

You're also given some starter code for the `Bvh` class. `Bvh.cpp` contains a simple import/export system for the Bvh, allowing you to save time by having to build a Bvh only once per scene. Additionally, `Bvh.hpp` contains some member variables for the class, most importantly the `indices_` vector. Instead of sorting the triangle list when building your hierarchy, you should sort the index list, through which your Bvh traversal code should access the triangles. This has the benefit of speeding up your hierarchy construction, as the amount of data you need to sort decreases, and it also means you won't have to save the whole sorted triangle list on disk in your hierarchy export code.

Use the spatial median or object median to determine the split based on the triangle centroids – specifically note that the spatial median split plane should be based on an AABB spanned by the *centroids* instead of the actual AABB of the node to ensure an actual split. C++ STL algorithms, especially `partition` and/or `sort`, along with lambdas, may come in handy when splitting the triangle list.

You will replace the brute force ray intersection routine in `RayTracer::rayCast` with a ray traversal that uses the BVH. In the leaf nodes, you can use the existing implementation to intersect against the few remaining triangles. Use the simple slab intersection methods described in the lecture slides, and pay attention to special cases, such as when the ray origin is inside a given bounding box. Notice also that in our implementation the ray direction R_d is not a unit vector. Rather, its length determines the desired maximum distance of the intersections, so that the intersections will be within the line segment $[R_o, R_o + R_d]$, where R_o is the ray origin.

After implementing all of this correctly, the performance of your tracer should be relatively close to that of the example – at least in the same order of magnitude. **Note that if you stick to the given format for your BVH, you can test your hierarchy construction and traversal separately by loading hierarchies generated by the solution code into your own ray tracer, and vice versa**.

R2 BVH saving and loading (1p)

A beginning note: this is way more important than it might seem – the rest of the course will be way smoother if you don't have to wait several seconds every time you want to test your renderer. Also note that if you stick to the given BVH format in R1, you will get this requirement for free. If you do customize the format, you will need to implement saving and loading functionality to get points for this requirement. Your custom format does not need to be compatible with hierarchies generated by the example solution, although compatibility will make it easier for yourself to ensure that your code works.

Building a good BVH is a slow operation. It makes no sense to wait for it every time since it can be serialized (saved) so easily. Implement functions `RayTracer::loadHierarchy` and `RayTracer::saveHierarchy` to cache the BVH. When you are ready to test your implementation, enable `bool tryLoadHierarchy` in file `App.cpp`.

You cannot save pointers in files. Reading a pointer from a file and accessing it will not work since it will point to the memory address where the data was located last time the application was run, not where it is located this time. This is a reason to use triangle indices instead of pointers to triangles.

Be careful to add in all data that you need to reliably restore your BVH data structure from the file. Obviously you need to restore your data from the file in the same order in which you wrote it. To store arrays, consider saving the element count before the elements so that you know when to stop reading the element data. If you had a branch for deciding whether you need to save the children of a BVH node or not, be sure to add in the required information to take the same branch when you read the data.

You can open a file for reading or writing by constructing instances of `std::ifstream` or `std::ofstream` in binary mode (see the code below). These streams will be automatically closed when they get out of scope. You should pass them as references to the corresponding save and load functions of your BVH. You can use functions like the ones below to write integer or floating point types to the stream. Again, no pointers.

```
#include <fstream>

// Write a simple data type to a stream.
template<class T>
std::ostream& write(std::ostream& stream, const T& x) {
  return stream.write(reinterpret_cast<const char*>(&x), sizeof(x));
}

// Read a simple data type from a stream.
template<class T>
std::istream& read(std::istream& os, T& x) {
  return os.read(reinterpret_cast<char*>(&x), sizeof(x));
}

// Save a kitten.
void saveKitten(const char* filename) {
  int kitten = 3;
  std::ofstream outfile(filename, std::ios::binary);
  write(outfile, kitten);
}

// Load a truck.
void loadTruck(const char* filename) {
  float truck;
  std::ifstream infile(filename, std::ios::binary);
  read(infile, truck);
}
```

R3 Simple texturing (1p)

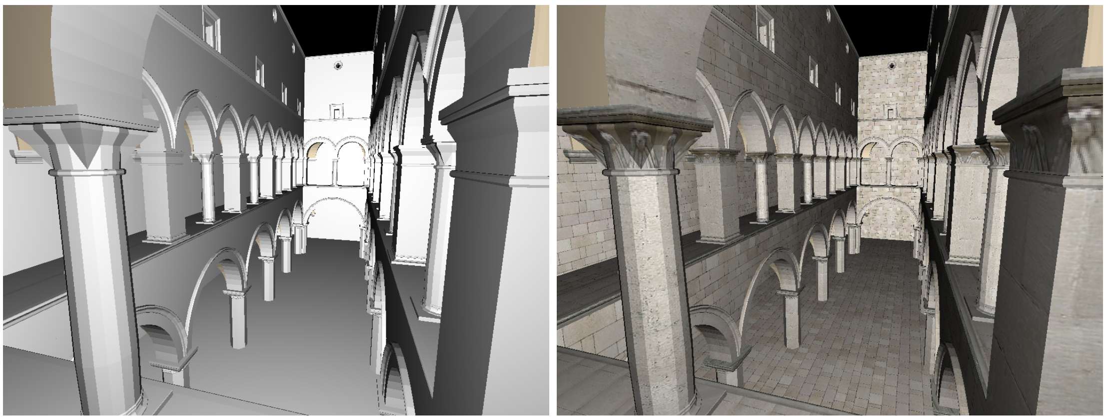
Texturing is an easy way to make rendered images more interesting without adding too much computational overhead. You're hopefully already familiar with the idea of wrapping an image over a mesh and using it to spatially vary a material parameter. The triangle structure already holds information about texture coordinates and the material that contains the actual texture. The ray-triangle intersection routine also returns the barycentric coordinates which are carried to the renderer in the `RaytraceResult` structure.

The barycentric interpolation of a vertex attribute t is defined as

$$t(\alpha, \beta) := (1 - \alpha - \beta)t_1 + \alpha t_2 + \beta t_3,$$

where α, β are the barycentrics and t_i the attribute at the vertices of the triangle (in this case, texture coordinates).

With all the necessary information already available, the task here is to find the texel nearest to the intersection. This is done by interpolating the texture coordinates according to the barycentric coordinates and reading the corresponding texel from the image. You should implement the interpolation in `getTextureParameters` in `Renderer.cpp` and fetching the wrapped texel coordinate in `getTexelCoords` in `RayTracer.cpp`.



The impact of texturing in the sponza atrium. Note that the reference executable implements the normal mapping extra which makes the pillars look smoother than in this image. Your solution should look like this image for the requirements.

R4 Ambient occlusion (2p)

Ambient occlusion (AO) is a simple and very popular shading effect that has no strict physical basis in reality. One can roughly consider it as an approximation to illumination from a perfectly cloudy sky. It tends to give a smooth shading that nicely accentuates the shape and details of the geometry.

The idea is as follows. The renderer has shot a ray through a pixel, and found an intersection at some point in the model. We then need to determine what color the pixel should receive; this process is called shading. AO is a shading method that, roughly speaking, evaluates how much unblocked "skylight" the point receives from all directions around it. If the point is in a crevice (such as a room corner), it will be dark; if it is in an open, unoccluded space (such as the middle of a floor), it will be bright. In practice, this is done by shooting several (say, 16 or 256) rays from the point to random directions using some maximum hit distance, and computing the percentage of rays that were occluded (i.e. hit another surface).

Implementation

In practice, the shading will be implemented in the `computeShadingAmbientOcclusion` function (in `Renderer`), which receives pointers to all relevant information in its arguments. The information you will actually need is the surface hit point and the surface normal at that point. The normal can be computed from the triangle vertex coordinates using a cross product (see the implementation of `computeShadingHeadLight`). To ensure that the surface will be shaded correctly when viewed from either side, you should flip the normal if the the camera is in the backside of the triangle. You should also nudge the hit point very slightly (say, 0.001 units) towards the camera, because otherwise the rays leaving the point may hit the surface itself due to rounding errors.

Ray direction generation. The ray directions are easiest to generate in a local coordinate system, from which they can then be rotated into alignment with the surface normal. The procedure for generating a single suitably distributed random direction in local coordinates is as follows:

1. Pick a uniformly distributed random point in the 2-dimensional unit circle on the xy-plane. This is easiest to do using rejection sampling: repeatedly draw two random numbers x and y in the interval $[-1, 1]$, until you hit upon a pair that is inside the unit circle (i.e. $x^2 + y^2 \leq 1$); accept this sample.
2. Lift the point vertically onto the 3-dimensional unit sphere by setting $z := \sqrt{1 - x^2 - y^2}$.

The resulting unit vectors $[x\ y\ z]^\top$ will be distributed on the upper half of the unit sphere according to a cosine distribution (the reason we use this distribution will be explained in future lectures.)

You can use a `FW::Random` to get random numbers. Note that an individual `Random` is not thread-safe. To parallelize our loop in `Renderer::raytracePicture` safely using OpenMP, we create a `Random` inside the loop so that each thread has its own instance.

Rotation to normal direction. Finally, the direction must be rotated so that the rays will be distributed around the surface normal instead of the z-axis. You will implement the generation of the rotation matrix R in the function `formBasis` in `raytracer.cpp`. The basic idea is that R must transform the z-axis vector $[0\ 0\ 1]^\top$ into the surface normal vector N, and it must be a rotation matrix. The first requirement is easy to implement: if we put N into the third column of R, then clearly $R\,[0\ 0\ 1]^\top = N$, regardless of the other columns in R.

The other two columns of R must be unit length and perpendicular to N and each other, because otherwise R is not a rotation matrix. In other words, the columns of R must form an orthogonal basis. We can construct a unit vector T that is perpendicular to N by picking any vector Q that is not parallel to N, taking the cross product $Q \times N$, and normalizing. This is because the cross product always returns a vector that is perpendicular to both of the operands. One way to pick a Q that is guaranteedly never parallel with N is to first set $Q := N$ and then replace the element with the smallest absolute value with 1.

We now have two perpendicular vectors N and T. To get a third one that is perpendicular to these both, we simply take the cross product $B := N \times T$. The matrix R can now be built by stacking the three column vectors side by side, as $R = [T\ B\ N]$.

Putting it together. The ambient occlusion is now easily computed by repeating the following procedure `Renderer::m_aoNumRays` times. Pick a random direction, rotate it to the normal direction, and trace a ray from the surface point to this direction with maximum length `Renderer::m_aoRayLength`. Note that the trace function uses non-normalized vectors, so that the length of the ray vector itself determines the desired maximum distance. Hence you should trace rays `m_aoRayLength` $* \tilde{u}$, where $\tilde{u}$ is a unit direction.

The returned pixel color is the number of rays that returned no hits, divided by `m_aoNumRays`.

R5 Simple multithreading (1p)

The base code is designed to render individual scanlines in parallel on multiple processor cores by using OpenMP; this can dramatically speed up rendering on a multicore machine. Parallel rendering is initially disabled to prevent any surprises to you. To enable it, you need to uncomment the line `#pragma omp parallel for` inside `renderer.cpp`, go to `Project -> Properties -> Configuration -> C/C++ -> Language` and set OpenMP support on.

To get the point from this requirement, enable multithreading and make sure your raytracing code is thread-safe. (Any extra credit features you write do not need to be thread-safe, but if they are not, you have to clearly document that in your `README.txt` and provide a simple way to disable the unsafe features.)

3 Extra credit

Here are some ideas that might spice up your project. The amount of extra credit given will depend on the difficulty of the task and the quality of your implementation. Feel free to suggest your own extra credit ideas!

We always recommend some particular extras that we think would best round out your knowledge. Recommended extras are typically easy to add onto the existing code and do not require large changes in the architecture of the code.

3.1 Recommended

- **BVH Surface Area Heuristic (3p)**
  Implement the surface area heuristic in your BVH builder. Include a toggle to enable and disable the feature, and compare and report the rendering performance against the simpler method(s).

  The Surface Area Heuristic (SAH, see Goldsmith) is a tree quality descriptor for ray traversal; it gives an estimate of the expected time cost of traversing a random ray in the tree. SAH relies on the fact that if a random ray hit node i, the ray also intersects a node j contained within i with probability $P(j \mid i) = A_j / A_i$, where A_k is the surface area of the bounding box of node k. The SAH cost of a tree t is

  $$\mathrm{SAH}(t) = T_{\mathrm{node}} \sum_{k \in \mathrm{inner}(t)} \frac{A_k}{A_{\mathrm{root(t)}}} + T_{\mathrm{tri}} \sum_{k \in \mathrm{leaf}(t)} \frac{A_k}{A_{\mathrm{root(t)}}} N_k,$$

  where T_{node} is the time it takes to intersect the child AABBs of an inner node, T_{tri} is the duration of a triangle intersection, and N_k is the number of triangles in node k.

  Based on SAH, a score can be derived (see MacDonald) for deciding the split plane placement in BVH construction. This SAH time cost of traversing leaf and inner nodes are

  $$\begin{aligned}
  &\mathrm{SAH}_{\mathrm{leaf}}(i) = T_{\mathrm{tri}} N_i, \\
  &\mathrm{SAH}_{\mathrm{inner}}(i) = T_{\mathrm{node}} + T_{\mathrm{tri}} \frac{A_L}{A_i} N_L + T_{\mathrm{tri}} \frac{A_R}{A_i} N_R \\
  &\quad = T_{\mathrm{node}} + P(L \mid i)\, \mathrm{SAH}_{\mathrm{leaf}}(L) + P(R \mid i)\, \mathrm{SAH}_{\mathrm{leaf}}(R),
  \end{aligned}$$

  where L and R are the left and right children of i. Note that this formulation results in a greedy optimization process; further splits of the child nodes are not considered, we simply try to choose the best alternative for the current step. It does not actually optimize the global SAH that well but leads to trees with relatively high performance – it turns out (see Aila) that this process optimizes other relevant performance metrics more so than the SAH which it was derived from.

To use SAH while splitting, pick planes at uniform intervals from the extent of the parent AABB along all three axis. Compute the heuristic for each plane and choose the plane with the smallest cost to split along. If the leaf node cost is less than the cost of any of the planes, terminate the recursion and stop splitting.

Experimentation is key to finding the constants T_{node} and T_{tri}; these will depend in somewhat complex ways on your specific implementation, hardware, and the scene geometry. You can either measure or simply come up with educated guesses; using 1 for both is a reasonable starting point.

Instead of the criterion given by the heuristic, you can choose to stop splitting simply when the current node is smaller than some chosen N (try around 10 to 30); this often yields better performance once you find the node size that works well with the cache (and will thus be hardware dependent). Doing this, you can omit all of the other constants; T_{node}, T_{tri}, and A_i are independent of the plane so the inner node heuristic simplifies to

$$\text{SAH}^*(i) = A_L N_L + A_R N_R,$$

and the leaf node heuristic can simply be ignored.

- **Efficient SAH building (1p)**
  Optimize your SAH builder such that you can evaluate the heuristic for all reasonable partitions of the triangles relatively quickly. This is possible by turning the split search from an $O(n^2)$ problem to an $O(n)$ one by using dynamic programming. Note that you'll need to do some sorting, so $O(n)$ is not quite true for the entire thing. However, you should still be able to do the split search in $O(n)$. This should be quite simple to do once you figure out how to do it, but you'll gain a nice tracing performance boost from the improvement in tree optimization.

- **Optimize your tracer (1-3p)**
  Improve the performance of your tracer to roughly match (1p) or substantially exceed (3p) the reference solution. Using a more sophisticated BVH builder (such as SAH) is probably necessary. Some other ideas:

  - Early exits in tree traversal and intersection routines
  - Implement the tree traversal with a local stack instead of recursion
  - Choose the minimum triangle count of the leaf nodes on your BVH carefully
  - Think about memory coherency; chasing pointers takes time

  We will publish the fastest tracers along with the round 1 points. We'll use the combined build and tracing times spent in the `comparison.bat` test case as the performance metric.

3.2 Easy

- **Implement proper antialiasing for primary rays (1-2p)**
  Let's do it the proper way this time.

This means that each pixel has its own prefilter (you can use a box, Gaussian, or Mitchell-Netravali bicubic filter sitting at its center. Normally the filters of neighboring pixels overlap (this is controlled by the standard deviation of the Gaussian or the width of the M-N bicubic).

Draw N random samples uniformly from each pixel and evaluate the color. When you've computed the RGB color $C = (C_r, C_g, C_b)$ for a sample at screen position (x, y), you then loop over all the pixels i whose filter overlaps (x, y). For this, you must know the width of the filter, in pixel units. (Note that this is particularly simple for a one-pixel non-overlapping box filter.) Then, you evaluate the filter $f_i(x, y)$ of the i'th pixel to get a weight w_i. Then you accumulate $w_i \cdot (C_r, C_g, C_b, 1)$ to the i'th pixel in the frame buffer (note that the fourth channel keeps track of the total weight accumulated in this pixel).

Once all samples have been drawn and processed this way, you loop over all the pixels and divide by the accumulated weight in the fourth component. Note that you must initialize the fourth channel to zero in the beginning before you start rendering for this to work (starter code doesn't do this, it just overwrites everything).

Note that if you take multiple primary samples per pixel, you will want to decrease the number of AO rays per primary ray to maintain the same number of AO rays per pixel.

- **Better sampling by low discrepancy sequences (1-4p)**
  In the previous extra you took N samples that were distributed uniformly inside each pixel. This is an example of stratified sampling over the image. In stratified sampling the space of random variables (e.g. ray positions) is divided into cells ("strata") and sampling is done uniformly from these cells. This is good since the distribution of these random samples tends to be more even and this results in less noise.

  You can improve by doing stratified sampling inside the pixels. Divide the pixels into cells and draw a sample uniformly from each cell. Another way is to use a low discrepancy sequence for the sample positions in the image (see `http://cg.informatik.uni-freiburg.de/course_notes/graphics2_04_sampling.pdf`). If you do this, we suggest that you implement the sample generators in a way that allows you to visualize the results directly, e.g., in Matlab, or a special mode you build in to the viewer itself.

  In addition to improving the sampling positions, you can also improve ambient occlusion by using e.g stratified sampling or a low discrepancy sequence for the ambient occlusion test rays.

Sobol samples on the left, uniform random samples on the right.

If you perform stratified sampling, you get 1 point. Designing your pixel antialiasing and AO sampling to jointly co-operate on a low-discrepancy sequence will earn 4.

- **Alpha and specular textures (1p each)** Alpha texturing makes it possible to cut out parts of triangles to introduce additional geometric detail without increasing the triangle count of the scene. In practice it means skipping the intersection depending on the value of the alpha texture at the intersection - for example if the alpha is less than 0.5 or less than a uniform random value from the range $[0, 1]$.

  For specular shading you can implement any additional material model that supports specular reflections and use the supplied specular texture to choose some parameter of the model, such as specular color or glossiness.

  Note that in our scenes, alpha and specular textures are only available in crytek sponza. You will need to play around with the scene's `.mtl`-file to load the additional textures. Have a look at the file with a text editor and find the corresponding textures to add from the scenes folder. This should not be too hard if you study how the diffuse textures are loaded.

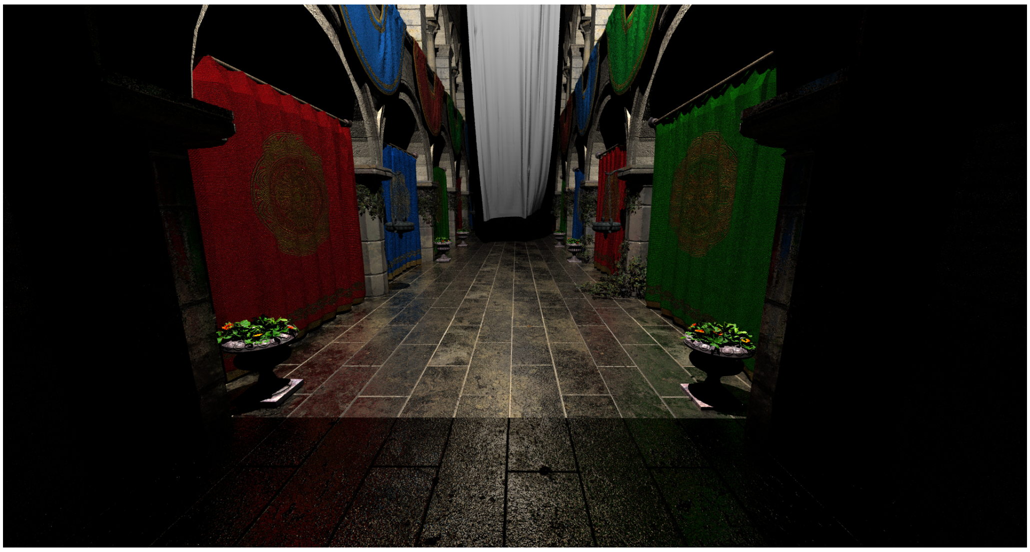
- **Implement and benchmark other bounding volumes (2p each)** Build BVHs from other primitives, such as spheres and/or oriented bounding boxes, and compare ray tracing performance between them. The more in-depth your analysis of the results, the more points you will receive on top of basic reporting.

3.3 Medium

- **Scene file system for multiple objects (3p or more)** Design and implement a scene file loader that allows for more than one object to be rendered in a single scene. For the simplest case, you should support rotation, translation and scaling separately for each object, as well as some ray intersection routine that takes these transformations into account. For further points, you can add support for object motion, stochastic motion blur, skeletal animation or anything else that you can think of. Further points will be awarded based on the sophistication of your solution.

- **Tangent space normal mapping (3p)** Normal mapping adds high precision detail to the scene with little additional cost and therefore allows you to render much more complex-appearing scenes in equal time. Crytek Sponza has tangent space normal maps which could be utilized. The easiest way to do this is calculate the tangent and bitangent on the fly after each ray intersection. The intuitive meaning of the tangent and bitangent is as the world space unit vectors that point in the directions where u and v increase. The formulas, more information and code snippets here.

- **Whitted integrator (2p)** Support perfectly specular reflections and point lights. The scene can be assumed to be of a diffuse material. Goes great with the texture extras.



Whitted integrator with normal and specular maps in the Crytek Sponza scene.

- **Ray differentials (2-3p)** When filtering a texture, it is important to know what the size of the filter should be. One good way of estimating the size is to find out the size of the pixel (or sample filter) on the texture to be sampled. This can be achieved by keeping track of so called ray differentials; helper rays from neighboring screen pixels that are assumed to intersect the same geometry as the main ray but at slightly different positions and angles, giving an estimate of the area that projects onto the original pixel. Two points for primary hits and another for reflective/refractive hits in the Whitted integrator. More details here.

- **Texture filtering(1-5p)**

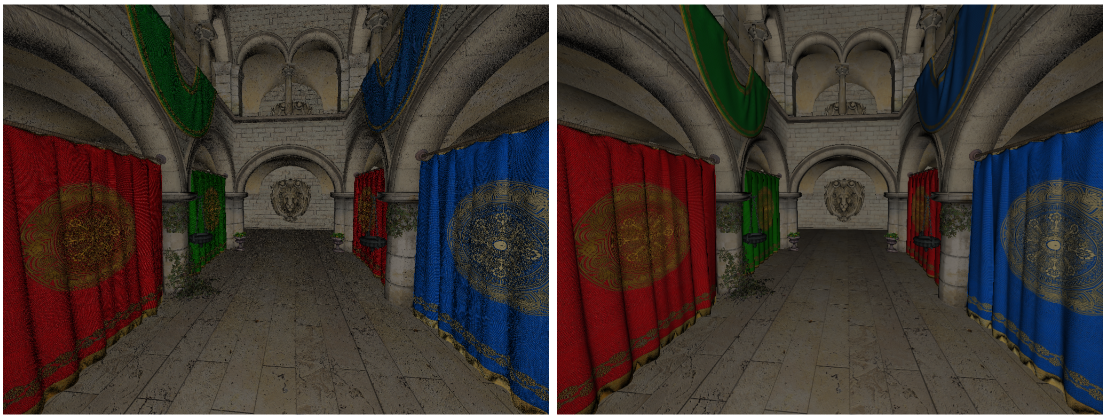
  Filtering textures is important to reduce aliasing of the final image, especially when taking only a few samples per pixel. The following techniques are increasingly good at filtering out high frequency content without excessive blurring of the image. They should be at least somewhat familiar from the Fall course, more pointers on the respective lecture slides.

For best results in the filters that use MIP maps, you should also implement Ray differentials to be able to properly approximate the filter size (i.e. MIP level). The paper linked in the Ray differentials extra gives the formula for the MIP level to use. It is also okay to do this extra on its own and come up with some other sensible formula for the filter size.

To enable MIP map generation in the mesh loader, go to the end of `parseTexture` in `framework/io/MeshWavefrontIO.cpp` and change `Texture::import` to `Texture::importWithMipMaps`. Then you can fetch a MIP level using the `getMipLevel` method of the `Texture` class.

Points awarded according to the hardest one implemented:

- Bilinear filtering(Easy): 1p
- Isotropic trilinear filtering(Medium): 2p
- FELINE filtering(Hard): 4p
- EWA filtering(Hard): 5p



No filtering on the left, trilinear on the right. Note the cleaner appearance of the cloth and tile details, especially in the distance.

- **BVH visualization (2-7p)**
  Using OpenGL, write code that visualizes the process of constructing your BVH, the traversal of a single ray in the BVH, or both. The points you'll receive will depend on how informative your visualizations are.

  For an example of how you could visualize the construction process: for each recursive call to the construction function, you briefly visualize the triangles currently under consideration, then visualize how they are split into left and right subtrees by coloring the triangles, and finally indicate which triangles go into a leaf being created. You'll probably want to draw the entire scene in some translucent color so that the triangles indicated as described above have some context.

  For visualizing the traversal of a single ray in the BVH, it might be most informative to use a 2D top-down view.

  Keep this in mind as a guideline: a good visualization is one that you can capture into a video and show on the lecture as a teaching tool.

- **Linear BVH construction using Morton codes (5p)**
  Add an alternative way to build a BVH using Morton codes as seen on the lecture. For details, see Tero Karras' blog post here. For this extra you don't need to perform the construction in parallel or on the GPU. Those things make excellent hard extras, though!

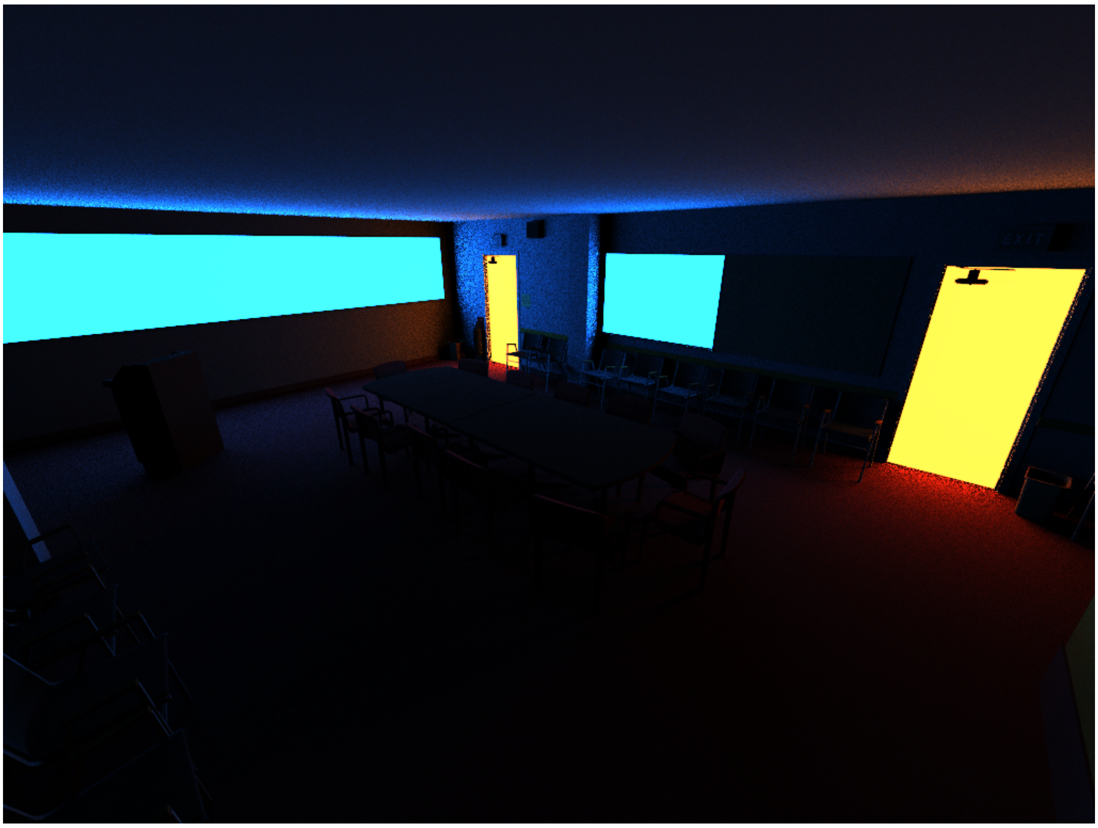
- **Constructing a clip space BVH (3p)**
  In order to optimize camera rays even further, you can transform all of the geometry into the clip space of the camera before constructing the BVH. Doing this

requires you to construct the hierarchy whenever tracing a new image after the camera has changed but will give you a performance boost. You can try keeping the AABBs in world space and optimizing based on the projected area (as in 4.8.2 in Havran's PhD thesis) or alternatively build the BVH and generate your rays all in the clip space.

- **Area lights with soft shadows (2-3p)** Implement a new shading mode that uses area lights. You should generate random points on the surface of the light and cast visibility ray for each point. If visible, add the incident illumination as described in the lecture slides. The starter code already generates a list of all emissive triangles in the mesh which you can use to pick a random triangle and draw a sample from its surface as described above.

  Note that when calculating the probability of the sample, you should divide by the number of triangles and the area of the picked triangle. You can also try to sample uniformly according to the combined surface area of all emissive triangles so the probability will be one over the combined area. It's also fine to come up with your own geometry for lighting, for example place a single emissive triangle in the scene and only sample that.

  Two points for a working implementation and one more if you support uniform sampling on the combined surface area of several emissive primitives.



Area lights in the Conference scene. The Cornell Box also has emissive materials.

3.4 Hard

- **Parallelize your ray tracer using SIMD (5-10p)**
  Modify your ray tracer to trace multiple rays at once using the 4- or 8-wide SIMD registers available in modern CPUs. You may want to look into Matt Pharr's ISPC compiler (http://ispc.github.io/) for an easy way of writing data-parallel code without the agonizing pain of using the assembly intrinsics. Since you only trace a small number of rays at a time, you can make use of this in the ambient occlusion rendering loop without re-engineering the scanline loop. **(5p)**

  In practice, it is best to re-engineer the rendering loop into two interleaved stages: first, you generate a number of rays into a batch, then you trace the batch all at once, after which another pass over the results updates the associated pixels. This is sometimes called "wave-based" ray tracing. Fortunately, the current case is simpler than the fully general recursive path tracer, because each primary ray shoots a fixed number of shadow rays in the ambient occlusion renderer. **Re-engineering your renderer to be wave-based, such that you process large batches of rays at a time, gets you 5 more points.** This paper will be helpful: http://research.nvidia.com/sites/default/files/publications/laine2013hpg_paper.pdf

- **Write a "software" GPU ray tracer (10p)**
  The "software" here refers to general-purpose code that is run on the GPU and written using CUDA, OpenCL, or the like — not making use of the hardware ray tracing features of modern GPUs. This is an even larger step, which requires you to trace even larger batches of rays at a time. It is crucial that your renderer is wave-based (see above). See Aila and Laine for tips on how to do this efficiently (paper at http://research.nvidia.com/sites/default/files/publications/aila2009hpg_paper.pdf, code at https://code.google.com/archive/p/understanding-the-efficiency-of- – and do not hesitate to ask questions!

- **DXR or Vulkan ray tracing (open-ended)**
  Create a version of this assignment that makes use of hardware ray tracing through the Vulkan or DXR ray tracing APIs. This is a very large undertaking that requires you to switch graphics APIs altogether, and should not be approached lightly. Significant prior experience in modern graphics APIs (Vulkan, DirectX 12) is recommended.

4 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Windows with Visual Studio 2022, preferably in the VDI VMs. You may develop on your own platform, but the TAs will grade the solutions on Windows.
- Comment out any functionality that is so buggy it would prevent us seeing the good parts. Use the `reference.exe` to your advantage: does your solution look the same? Remember that crashing code in Debug mode is a sign of problems.
- Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code. Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, project and solution files required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. *Do not* include your `build` subdirectory or the reference executable. Beware of large hidden `.git`-folders if you use version control - those should not be included.
- Sanity check: look inside your ZIP archive. Are the files there? Test and see that it actually works: unpack the archive into another folder and see that you can still compile and run the code.
- If you experience a system failure that leaves you unable to submit your work to MyCourses on time, prepare the submission package as usual, then compute its MD5 hash, and email the hash to us at `cs-e5520@aalto.fi` before the deadline. Then make the package that matches the MD5 hash available to us e.g. using `filesender.funet.fi`. *This is only to be done when truly necessary; it's extra work for the TAs.*

Submit your archive in MyCourses in "Assignment 1" by February 16, 2025 at 23:59