Final Project - Part 2

This second part of the project will be performed in groups of three. When you read this document, you should have been assigned to a group already. Please get in contact with your group members to discuss your planning before next week Wednesday 14th of October. If you do not, your group members can inform us that you are inactive and we might remove you from the project. Similarly, use the time to make a rough planning on who is working on what part. If you cannot agree, please inform us before this date. This project part accounts for 70% of the final project grade - 30% were already obtained with the individual elements (Part 1 + Blender).

Before we start, some important general points to keep in mind:

- Copying code is fraud if it might look as if it is your own. This includes code from others or internet sources and holds even if you modify the code or translate it from another language.
- All external resources should be clearly identified in the reportand your code (in the form of comments). Please provide specific links to the websites, YouTube videos, book chapters, articles,etc that you use (e.g., “wikipedia” would not be a valid reference for the Wikipedia page of the Phong Model, you should include the link). The use of external resources, if containing code, might result in a deduction.
- Only one common submission is expected per group but the grades can be individual. We will rely on your work distribution overview. If the differences do not exceed 15% in terms of the associated points, we will consider that the workload was well-balanced and the members will receive the same grade. Otherwise, the grades will be adapted.
- There will be no presentation of your project butsome groups will be contacted for a post-submission interview. In the case you are interviewed, you can expect to receive an invitation email up to 10 days after you handed in your project. If you are invited, your grade will be withheld until after the meeting. The interview will be scheduled together to make sure everyone is available.
- Each group member should be able to explain any part of the code that they were involved in. For example, if you contributed to “soft shadows”, you need to be able to explain all code related to “soft shadows”. Otherwise your breakdown needs to be more precise.
- Please use GitHub to create a repository. Please be careful tomake your project private! as it may otherwise be considered as “sharing code”. You have to make this repository also available to cg-githubUser.
- The deadline is the 30th of October 2020 at 23:59. Please upload early to avoid problems with Brightspace or the internet. You can upload multiple times but only your latest submission will be kept.

1. Basic description

In this assignment, you will build upon the intersection tests that you implemented in Part 1 by creating a fully functional ray-tracer.

The user interface enables you to choose from a selection of test scenes to work with, including a custom scene. It initially loads a scene custom.obj in the data folder. If you like to load your own scene, you can simply overwrite the corresponding custom.obj with your own scene (please note that you should not rename your obj material file (MTL), as the original filename is mentioned within the obj file itself).

The provided framework is similar to the code you worked with in Part 1, so you can import your intersection methods to the new code. Note that the sphere, triangle and mesh intersection methods now have an additional (output) argument `hitInfo`. `HitInfo` can be used to store information about an intersection point that can then be used for shading, such as the surface normal and the material. Please see the Appendix for a more detailed description about Materials. You can store this information after a successful intersection test.

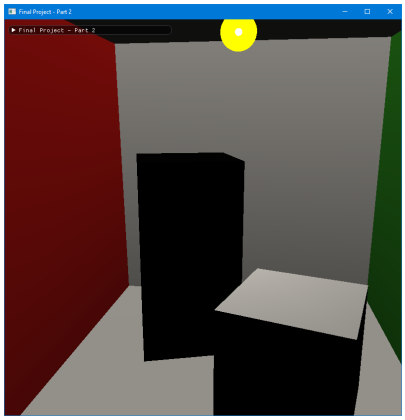
```
// Defined in ray tracing.h; You are free to change it and add information.
struct HitInfo {
    glm::vec3 normal;
    Material material;
};

bool intersectRayWithTriangle(const glm::vec3& v0, const glm::vec3& v1, const g
bool intersectRayWithShape(const Mesh& mesh, Ray& ray, HitInfo& hitInfo);
bool intersectRayWithShape(const Sphere& sphere, Ray& ray, HitInfo& hitInfo);
```

Compilation

For a start, you can compile the code and test the application. Please note that you can compile the code in two different modes “Debug” or “Release”. When compiled in “Debug” mode, there are a lot of additional tests that are done behind the scenes, e.g., if you are accessing a std::vector, it always checks that you stay within its bounds. It is very useful to compile in “Debug” mode when developing code to ensure that all these tests are performed. Nevertheless, once the code is working, you should switch to “Release”. If you now compile the code, it will be highly optimized and result in a tremendous speedup. Especially, when working with complicated scenes, it is a good idea to rely on “Release” mode. In the same spirit, when in “Debug” mode, you should rather work with simpler scenes to avoid long wait times. Whatever the mode, once compiled, we can launch the application.

Rasterization mode

From the menu, you can choose a “Rasterization” and a “Raytracing” view mode. The OpenGL Rasterization mode will be used for debugging and interaction. The “Ray traced” view mode will launch actual rays per pixel. You can also click the “Render to file” button to render to a file instead.



Screenshot of the raster view.

By pressing `r` when in “Rasterization” mode you can launch a single debug ray through the mouse cursor. Throughout the project, you will be asked to implement this debug functionality, as it is a useful way to illustrate how the ray progresses through the scene; you can visually inspect if reflections are going in the right direction and the correct material properties are recovered.

The visual debugger can be implemented in the same function (`getFinalColor`) as the rest of your ray tracer. For the exercises you should always include both the ray-tracing implementation, as well as the realization of a visual debugger. The latter is always the element to start with. By having visual feedback of your calculations, you are able to catch bugs early on in the development. Experiment with the debugging outputs. For example, you can color the ray to illustrate the current depth with respect to the number of reflections, or you can draw a line to show each normal of the surfaces that are encountered by the ray. You might notice quickly that these types of outputs are very useful when debugging your work. Not all functionality of the visual debugger has to be available at once during the execution of your program. You can comment your visual debugger code in and out depending on the part of the code that you work on and which you like to debug (alternatively, it is very easy to extend the interface by adding an option: `int v; ImGui::DragInt("Label", &v);`).

You can also insert, remove and modify light sources using the GUI in Rasterization mode. The light sources will be used for ray-tracing. But please note that the illumination in Raterization mode will not necessarily correspond to the illumination in Ray-Trace mode. Here, we just placed a single light source for the scene to have some illumination.

Ray-Trace Mode

To produce an actual ray-traced image, you switch to the “Ray Traced” view mode. Here, the program will generate one ray for each pixel on the screen. Ultimately, for each such pixel, you will compute a RGB colour value by determining the light that is reflected towards the camera pixel, taking reflections, shadows and other effects into account. This should be implemented in `getFinalColor(...)` in `main.cpp`.

If you press “Render to file” the image is ray traced and written to a file `render.bmp` in the root folder of the project. This functionality is also available in the rasterization mode which is useful if the ray traced mode is too slow. Make sure the path to this folder does not contain any special characters (only ascii characters).

For a simple scene, the Ray-trace mode might be interactive but as soon as the amount of geometry increases and the effects become more complex, the process will be costly. In consequence, you will also be asked to create a data-structure to accelerate your ray-tracer, as we will outline in detail below.

When a scene is loaded, it is centered and scaled to a `[-1,1]x[-1,1]x[-1,1]` volume. Hereby, the placing of the light sources in the scene via the interface is facilitated. Via the interface, you can add light sources, change their type/color or move them around.

Additional advice regarding the code and use of specific classes can be found in the appendix of this document.

2. Standard features

If you have completed the entire individual project successfully, you have already gathered three points of the final grade of the project. The minimal project grade to complete the course is a 5.0 (see the studyguide). Below, you will find a list of features with an indication of how many points they represent. You are free to work on other features but you have to first finish the standard features as outlined below.

Assuming that you have obtained 3 points from the individual assignment, each group member needs another 3 points to reach a 6.0. Below, we list the *standard features* whose points are shared among those that contributed to the realization. For example, if all three members implemented the recursive ray tracing (1.2 points total) together and contributed equally, each one would receive 0.4 points.

- [0.5] shading using Phong Illumination Model with visual debug function
- [1.5] recursive ray-tracer (reflection ray, shadow rays) with visual debug
- [1.0] hard shadows with visual debug
- [1.5] soft shadows with spherical lights with visual debug
- [6.0] acceleration data-structure without pointers and with visual debug
- [1.5] report including timings of your implementation

Note that each of these standard features consists of the actual feature to be implemented, as well as a visual debugger, whose importance was outlined above. The indicated points are the maximum, assuming that the implementation is entirely correct.

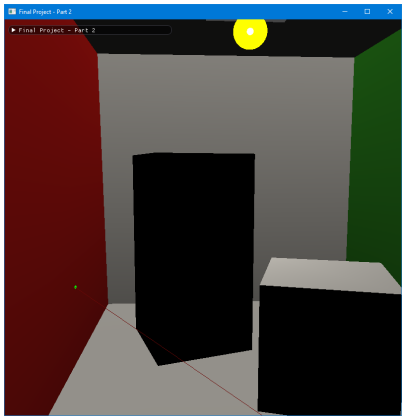
When testing your implementation, we will use Visual Studio on Windows to compile your work, as for the assignments. Please note that you should not (and do not have to) use any external libraries. If you do, please make sure that your project can be compiled without any extra efforts or it might affect the grading.

2.1 Shading with visual debugger

In `getFinalColor(...)`, at an intersection point, you should compute the direct illumination using the Phong Model for all light sources in the scene. You can access the light sources via `scene.pointLights` and the material parameters can be passed along by the intersection function using the `HitInfo` struct as described above.

Visual debug: draw the camera ray using the `drawRay(ray, color)` function that is defined in `draw.h`.

This function can be called from within `getFinalColor(...)`. Tip: call `drawRay(...)` after you intersect the ray so the ray has a length (`ray.t`).

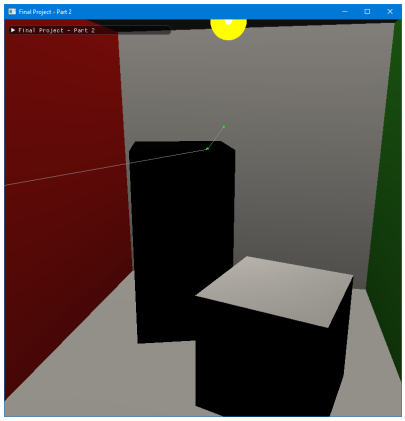


Example of what the visual debug might look like.

2.2 Recursive ray-tracer with visual debugger

In Part 1, you already implemented intersections. Now you need to recursively trace rays that intersect specular surfaces. Every time the ray intersects a specular surface (`material.ks` is not black), you should trace another ray in the mirror-reflection direction. The reflected ray will contribute additional light to the specular component of the illumination model.

Visual debug: extend upon the visual debug from Part 1 by also drawing the reflected rays. This should be combined with the visual debug from shading (above).

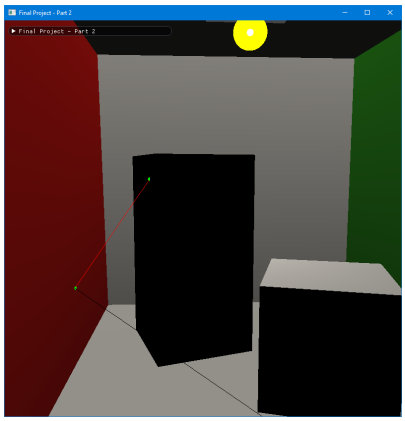


Example of what the visual debug might look like.

2.3 Hard shadows with visual debugger

For point light sources you should compute hard shadows. At each intersection point cast a shadow ray towards each light source to determine if the point is in shadow or not. If the point is in shadow for a light source, then that light source should not contribute to the shading computation.

Visual debug: draw the shadow ray. If the shadow ray is occluded (it hits something other than the light) then draw it in a different colour such as red. This debug display should be combined with the visual debug for shading and the recursive ray-tracer.

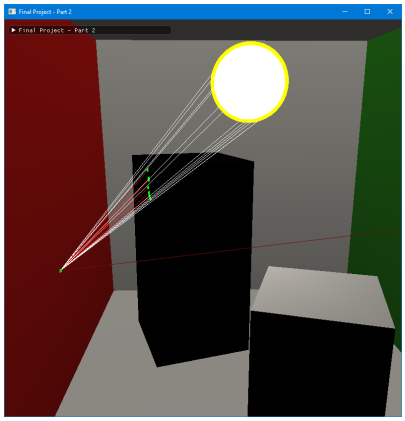


Example of what the visual debug might look like.

2.4 Soft shadows with spherical lights and visual debugger

Instead of a point light source, now you should create an area light source, more specifically, a spherical light. The spherical light source has a centre and radius. To compute soft shadows you should sample the sphere with shadow rays and determine the percentage of coverage of the light. The sampling method should be described in the report.

Visual debug: As with the hard shadows. Draw all the rays that you sample and color them according to whether they hit the light or not.



Example of what the visual debug might look like.

2.5 Acceleration data-structure without pointers and with visual debugger

This is the most laborious part of the final project and the requirement that will allow you to render large scenes in a reasonable amount of time. You should implement a Bounding Volume Hierarchy, or BVH for short. Please use Axis-Aligned Bounding Boxes as bounding volumes (as seen in the lecture) and a binary tree (every parent node has two children). The `AxisAlignedBox` structure is defined in the `scene.h` file. Initially, you should create an AABB that contains the entire scene, and then subdivide it creating two new child AABBs, and distribute the content among the children. You should then proceed to subdivide the child nodes recursively, constructing the hierarchy. Don't forget to limit the maximum number of levels of your hierarchy.

You can implement any subdivision criterion, but you should specify your choice in your report. For example, considering an AABB with a number of triangles inside, you could split it at the position of the median triangle at the *x-axis*, and alternate the splitting axis for the next subdivision levels. Another option is to implement SAH+binning, as seen in the lecture.

The bounding volume hierarchy only has to work for meshes, support for spheres is optional. Please organize your BVH such that you do not use pointers and instead rely on the standard std::vector class.

We have already set up the basic methods inside the`bounding_volume_hierarchy` files. Use these files to implement your BVH. Initially, there is no BVH, so the intersection method loops over all meshes to test

for intersection against all primitives. In the following, you will implement several methods in the BoundingVolumeHierarchy class to create this data structure. The following are the main methods, but you are allowed to create new methods as well:

- `Constructor`: receives as input the scene and you should implement the hierarchy construction
- `numLevels`: returns the number of levels of your data structure. Initially it is set to one, you should change it accordingly once the data-structure is created.
- `debugDraw`: receives as input the BVH level to be drawn, you should draw all bounding volumes (AABBs) at that hierarchy level (see Visual Debug below). A small example on how to draw a box is contained in the file, and more information in the Appendix.
- `Intersect`: once your hierarchy is constructed and you are able to visually debug it, you can finally implement the ray against BVH intersection test. This method receives a ray and the `HitInfo` to be filled. Here you should traverse the hierarchy and find the closest intersection point, if any, in order to fill the HitInfo structure. Note that for recursive ray-tracing, you might need to traverse the structure multiple times.

To construct the BVH you will need to create a `Node` struct in the `bounding_volume_hierarchy.h` file (you can look at the `scene.h` file for some struct examples). All the nodes are stored in a *node vector* of type `std::vector<Node>`. A node can be of two types: *interior* or *leaf* node. You can use a *boolean* to identify the Node type. Every node contains its own `std::vector` that can be used to store the indices of the two child nodes in the *node vector* (in case of a parent node) or store the indices of the contained triangles (in case of a leaf node).

Only after having finished the outlined BVH implementation, you could propose other data structures, such as Kd-trees, Octrees, regular grids - please discuss the implementation of these structures with R.Marroquim@tudelft.nl before starting.

Visual debug: visualize one level of the hierarchy at a time by drawing the respective AABBs. The GUI contains a button to enable/disable the BVH visualization for debugging. If enabled, an additional slider will become available to choose the level in the hierarchy. This value is passed as input to the `debugDraw` function.

3. Extra requirements (extra features)

You can implement many other features. For some, we list the corresponding points below but other ideas are welcome. If you are in doubt if an idea for an extra requirement is valid or “worth it”, please contact M.L.Molenaar@tudelft.nl or R.Marroquim@tudelft.nl. Most suggestions can be implemented with the material from the lectures but some might require extra research (we have marked those with *).

- [0.5] Interpolated values over the triangle via barycentric coordinates.
- [0.5] Soft shadows for other light sources, such as a planar area light.
- [1.2] Motion blur, explain your solution in the report.
- [1.0] Bloom filter on the final image.
- [0.5] Antialiasing: similar to mipmaps, reduce the resolution of the output, averaging 2x2 pixels.
- [1.2] Texture mapping (you can extend and use the Image class, which is provided in the code).
- [0.5] Texture filtering: Bilinear interpolation.
- [1.2] * Texture filtering: Mipmapping.
- [1.0] * Cast multiple rays per pixel. Describe your sampling strategy in the report.
- [1.5] * Glossy reflections. For glossy or diffuse surfaces cast recursive rays at directions different from the mirror reflection.
- [1.5] * Transparency, explain your solution in the report.
- [2.0] * Depth of field, explain your solution in the report.

Visual debug: whenever possible create a visual debugger for each extra feature and include screenshots of your debugger working.

4. Report

Your report is essential for the evaluation. Read the instructions carefully and include all the required points. We will only evaluate what you list in the report

Start the report with the group number and a list of all members (names and student numbers).

4.1 Work Distribution

At the beginning of the report, you should have a table with a group member per column, and each implemented feature per row. For each cell indicate the percentage of work done by each member for each feature. Example:

Feature	Student 1	Student 2	Student 3
Shading	50%	25%	25%
Recursion	40%	30%	30%
Data Structure	10%	10%	80%
Hard Shadows	20%	80%	0%
Soft Shadows	20%	60%	20%
Transparency	100%	0%	0%
Compact BVH	0%	0%	100%
Interpolated Normals	0%	70%	30%
Textures	80%	20%	0%

4.2 Features

For each feature, you must include:

- A short description of the implementation. Do not copy-paste code but describe what you did in a few sentences. Add required details, such as subdivision criterion for BVH or the sampling method for the spherical light source.
- At least one rendered image that properly illustrates this feature. From the image, the feature has to be clearly identifiable. For example, a render with soft-shadows does not also count for hard shadows. Here, two images should be generated, one image using a point light and another with a spherical light source. The same reasoning holds for all other features.
- The visual debugger (if implemented for a feature) needs to be shown in an image or sequence of images to illustrate its use. This can be a screenshot from the OpenGL window. For example, for the data-structure, illustrate the different levels with multiple images.

>>If you do not include these elements, you might not receive all points for a feature<<

4.3 Models

You can use your own models to produce images. Be sure to specify which model belongs to which group member. If a scene is not modelled by one of the group members, make sure to include a reference from where the scene was taken. As mentioned above, you can overwrite the file custom.obj in the data folder with your own model and it becomes directly accessible from the interface by selecting the custom scene.

4.4 Performance test

Please provide a table with all scenes (even those containing only one model) that you rendered. It should contain the scene name, number of triangles, time required for rendering. In case you have implemented any additional effects, you do not have to use them for the timings. You can limit the number of maximum recursions to three. Please report on the number of subdivisions that your data structure used.

For each scene make sure that you have included an image as requested above.

Example table (these numbers are fictional, do not use them as a reference):

	Cornell Box	Monkey	Dragon	Blender Stud 1	Blender Stud 2	Blender Stud 3
Num triangles	32	968	87K	10K	120K	22K
Time	0.5s	2s	120s	20s	210s	35s

	Cornell Box	Monkey	Dragon	Blender Stud 1	Blender Stud 2	Blender Stud 3
BVH levels	3	5	9	7	10	8

You can also add more tables with performance measures, such as to illustrate the cost of an extra feature, or how rendering the same model with different amounts of subdivisions for the data structure affects the render time.

5. Image competition

Submit your best image to the render competition. Just include an image file in your zip folder with the name *competition-groupXX.png*.

With this competition, you can gain up to 1.5 points for your final project grade (limited to 10). You can use any model you want but do not forget to include references. The ranking will be based on a combination of technical merit and visual quality.

6. Submission

You should submit all your source code files (not the content of the directories “debug” and “release”) in a zip folder, including your PDF report named (“report-groupXX.pdf”). You should also submit all scenes that you have used in your work. Please upload early, but if you encounter difficulties with the upload, calculate an md5 checksum (e.g., https://www.winmd5.com/) of your zip file and send it to M.L.Molenaar@tudelft.nl or R.Marroquim@tudelft.nl before the deadline, indicating your group number.

Please make sure that your code compiles when you submit it. We will use Visual Studio on Windows to compile your work, as for the assignments.

7. APPENDIX

7.1. Materials (MTL)

OBJ files have an optional accompanying MTL file. When you create a model with materials in Blender, the MTL file is also exported together with the OBJ file. Note some advanced material options in Blender are not supported by the MTL file format.

The MTL files are automatically loaded by the assimp library included in the framework, since the MTL filename is mentioned in the OBJ file. We have added a Material structure to the framework to allow easy access to the Phong coefficients. The following material structure can be found in the `mesh.h` file:

```
// Defined in mesh.h
struct Material {
    glm::vec3 kd;
    glm::vec3 ks { 0.0f };
    float shininess { 1.0f };
    float transparency { 1.0f };
};
```

For example to access the diffuse coefficient of a mesh use:

```
glm::vec3 = mesh.material.ks;
```

We have added one additional parameter to the material structure, `transparency` with values ranging from 0.0 (fully transparent) to 1.0 (fully opaque). This is only to be used if you do implement transparency as an extra feature, otherwise you can safely ignore this parameter. In Blender you can set the transparency by going to the Material tab, activating transparency and setting the Alpha value. This value will be exported to the MTL file and imported by the framework.

If you created a model that uses multiple materials, they will be automatically handled. The `loadMesh` method in `mesh.cpp` actually returns a vector of meshes, and not a single mesh, hence, each submesh has its own material properties.

Remember that the initial BVH structure already loops over all submeshes for you. So for each submesh you can directly access its material object.

7.2. Drawing rays

A ray is defined by its origin, direction, and intersection parameter *t*, as defined in Part 1. You can define your own rays for debugging and call the method `drawRay` to render it in Rasterization mode. This function also draws a small sphere at the intersection point, unless you do not modify the *t* value (it is set to infinity - a very large value). You can also come up with your own draw functions if you like.

For debugging purposes, a global variable `enableDrawRay` controls if the rays are rendered or not, so make sure it is set to *true* before rendering your rays. If you draw a ray inside the `getFinalColor` method then the flag will already be set to *true*.

Here is an example of how to draw a green ray:

```
Ray ray;
ray.origin = glm::vec3(1.0, 0.0, 2.0);
ray.direction = glm::vec3(0.0, 0.0, 1.0);
ray.t = 3.0;
glm::vec3 colour (0.0, 1.0, 0.0);

// Declared in the `draw.h` file
drawRay(ray, colour);
```

7.3 Drawing boxes

In the `draw.cpp` file you will find a method to draw Axis Aligned Bounding Boxes (AABBs):

```
void drawAABB(const AxisAlignedBox& box, DrawMode drawMode, const glm::vec3& co
```

The draw mode can be either `DrawMode::Filled` to render the sides of the AABB, or `DrawMode::Wireframe` to render only the edges. You can also pass a color and transparency value. This last value can be used with DrawMode::Filled to draw transparent boxes, where 0.0 if fully transparent and 1.0 is fully opaque.