

Lab2 – Ray Tracing

If you are doing this exercise in room 5355, the information on **this** page applies – otherwise not.

DTEK-students should preferably use their own accounts. Other students can get a temporary account from me.

Due to problems with the computer system, there are some quick changes:

You will *probably* work with Visual Studio.Net instead of the older Visual C++.

Easiest way to start Visual C++ is by double-clicking the downloaded RayTracer.dsw file. Visual Studio.Net will *probably* start to install.

A window with 3 alternatives should appear (VB, C#, C++). Select Visual C++. There will *probably* pop up at least two error messages. Ignore them by selecting appropriate YES/NO answers...

To build – press **ctrl + shift + B**, or select menu Build->build or Build->rebuild all

Unfortunately, running or debugging probably still does not work from within Visual C++ due to access rights. MEDIC has informed me that this hopefully will be solved before week 5. To run, compile the program and copy the file from the Debug-directory one step up to the same directory as the RayTracer.dsw-file. Run from there, by for instance double-clicking on it. This procedure unfortunately has to be repeated each time you recompile and want to run the program...

Lab-PM: Lab2 – Ray Tracing

In this exercise we will study ray tracing.

Documentation of the file format (nff – “Neutral File Format”) that we are using for the scenes, can be found online at:

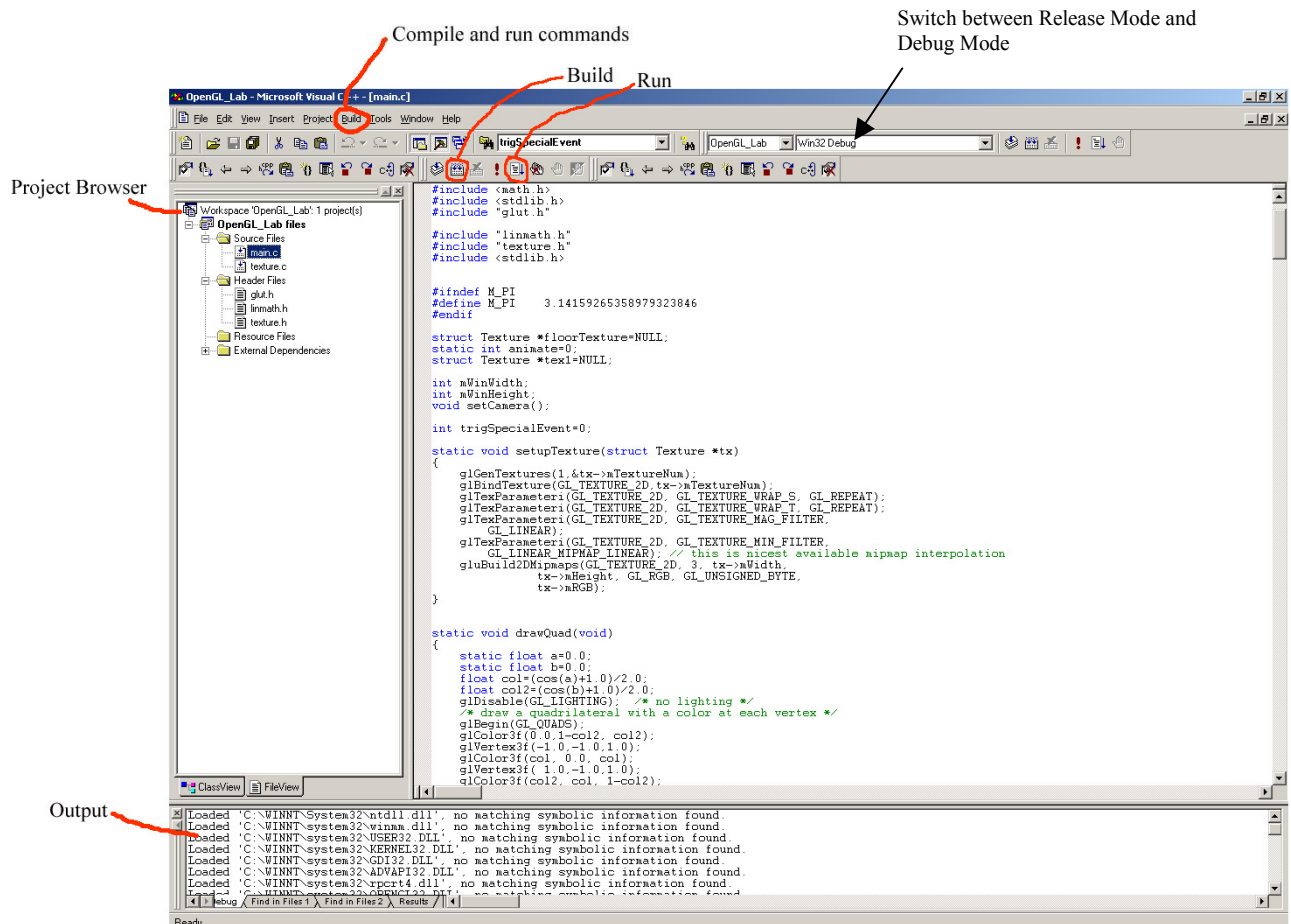
<http://www.acm.org/tog/resources/SPD/NFF.TXT>

Set-up

1. Download the file lab2.zip from this course’s home page.
<http://www.ce.chalmers.se/undergraduate/D/EDA425/labbar/minilabbar.html> .
There are two similar versions of the LabPM – one for Word and one as pdf, in case one of Acroread or Microsoft Word is missing on your computer.
2. Unzip lab2.zip.
3. Double-click on the file RayTracer.dsw to start Microsoft Visual C++.
Note that the code is written in C, and not C++. This is controlled by the file extension (c vs. cpp).

A Quick introduction to Microsoft Visual C++

We will use Microsoft Developer Studio in these exercises. If you want to use another compiler, you may do so, but then we probably cannot provide any help regarding compiling and linking.



Project Browser: This browser shows all the files in the project. Double click a file to open it for editing. You can use the **Window**-menu to watch several files simultaneously.

Build Menu: Commands for building (compiling and linking) the project and running the program. There are also two toolbar buttons for this.

Output Window: This window shows the output of the compiler, linker, and etcetera. It does, however, not show the output of the program.

For Help on OpenGL commands: Use menu **H**elp->**I**ndex. Type for instance **glEnable** as keyword.

Switching between Release Mode and Debug Mode. Release mode optimizes the code, and thus makes the ray tracer faster.

Ray Tracing

1. Startup

Notice the structure of the program.

To display the scene in a window, we use OpenGL and GLUT.

The related functions, which reside in file main.c, are: **main()**, **display()**, and **reshape()**.

In **main()**, the scene is read from an input file with the command:

ReadScene(&mScene, argv[1]);

argv[1] contains the file name. In this case the file "balls1.nff". In Visual C++, the command line arguments are set by the menu option: Project->Settings-> "tab Debug" -> "Program Arguments".

The ray tracing functions in file main.c are:

1. void **trace**(int level, float weight, Ray3f ray, Vec4f color);
Traces one ray through the scene and returns the color at the hit point.
2. void **tracePixelPos**(float i, float j, Vec3f xstep, Vec3f ystep, Vec3f vecToLowerLeftCorner, Ray3f ray, Vec4f color)
Similar to the above function, but this one takes the pixel coordinates (i,j) as input and calls **trace()**.
3. int **Shadow**(Ray3f shadowRay)
Determines if the shadow ray hits any object. Returns true, if any object is intersected.
4. void **shade**(int level, float weight, Vec3f P, Vec3f N, Vec3f I, Material mtrl, Vec4f color)
Computes the shading at point P. P = surface point, N = surface normal, I = incident ray (swedish: infallande stråle), mtrl = surface material parameters, color = returned color at P.
5. void **screen**(void)
Traces a ray through each pixel of the screen and creates an image stored in "mPixels".

The ray tracing functions in file intersect.c are:

1. int **Intersect**(Ray3f ray, Scene* mScene, Geometry** outObject, Vec3f P, Vec3f N, Material** mtrl)
Master intersection function that calls the other below. Returns 1 if an object is intersected. OutObject = the intersected object, P = the intersection point, N = the normal of the intersected object at the intersection point, mtrl = the material for the intersected object.
2. void **SphereIntersect**(Ray3f ray, Sphere* sphere, int* bHit, float* t, Vec3f P, Vec3f N, Material** mtrl)
Computes the closest intersection point between a sphere and a ray
ray = the ray to be tested, sphere = the sphere to be tested, bHit = (returned) non-zero if the ray intersects the sphere, t = (returned) distance from the ray

origin to the closest intersection point, P= (returned) the closest intersection point at the sphere, N= (returned) the surface normal at point P, mtrl= (returned) material parameters at point P

3. void **CylinderIntersect**(Ray3f ray, Cylinder* cyl, int* bHit, float* t, Vec3f P, Vec3f N, Material** mtrl)

This one is to be filled in by you in the last exercise, if there is time.

4. void **PolygonIntersect**(Ray3f ray, Poly* pPoly, int* bHit, float* t, Vec3f P, Vec3f N, Material** mtrl)
Computes the intersection between a ray and a polygon, with parameters as for SphereIntersect().

Study the file types.h. Notice the type definitions of struct **Geometry, Scene, Material, Cylinder, Sphere, Poly, Light, LightList** .

Study file linmath.h. This file contains definitions for several vector functions, like dot products, cross products, add, sub, scale, normalizing, and set.

Compile the project OpenGL_Lab (with button **F7**) and run the program (with button **F5**). You should see an orange polygon and some spheres on a blue background.

1. Exercises

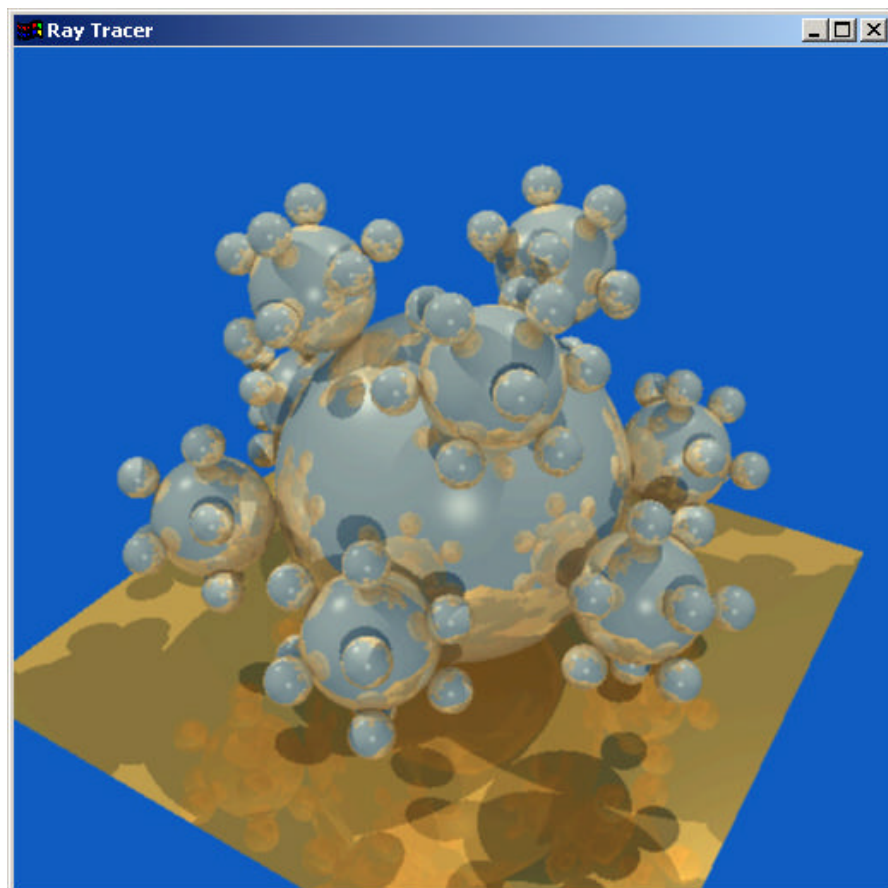
The image looks a bit weird, and we will start by fixing that. because depth sorting is not done, and shadows are not computed.

1. First, we will correct the depth sorting. Currently, the ray tracer only searches for the first intersection between the rays and the objects. Correct this by modifying function int **Intersect**(Ray3f ray, Scene* mScene, Geometry** outObject, Vec3f P, Vec3f N, Material** mtrl) in file intersect.c. The function should search all objects and return the one closest to the ray origin.
2. We should now add shadows to the image. Fill in the function **Shadow**(Ray3f shadowRay). The function should return 1 if any object is intersected. Otherwise it should return 0. Note, that this is very easy if you use one of the above functions.
3. Add specularity to the shading. In function **shade**(), only ambient and diffuse shading is computed. Add specular shading at the line “// Compute specular contribution here”. It should be near line 74 in file main.c. **Hint**: you may use the simplified specular shading described at page 77 in Real-Time Rendering. The shininess is defined by the material parameter mtrl.phong_pow, which usually has a value between 1 and 128. OpenGL limits the value to 128, but this ray tracer does not. The parameter mtrl.phong_pow is already scaled with a factor 4, so if you do not use the simplified shading, you should divide the value with 4. Usable functions could be V3DOT, V3NORMALIZE, V3SUB, and pow(). You can define a vector of 3 floats with the type Vec3f.
4. Enjoy the enhanced ray tracer with the input file balls2.nff. The input file can be changed with the menu option: **Project->Settings-> “tab Debug” -> “Program Arguments”**. The image will probably take about one minute to render.
5. To know that your updated raytracer is correct, the produced image for balls2.nff, with the highlights, should look exactly as in the figure below.

6. If there is time, extend the ray tracer to handle cylinders. Fill in function `CylinderIntersect()`, in file `intersect.c`, to compute the intersection between a ray and a cylinder. The intersection point **P**, the normal **N** of the cylinder at the intersection point, and the distance **t** from the ray origin to P should be computed. The material **mtrl** of the cylinder should also be returned. ***bHit** should be 1 if there is an intersection, and 0 otherwise.

Test the intersection routine with the file `cylinder1.nff`. When everything seems to work, use file `cylinder2.nff` instead.

7. If you have time and a fast computer, test the ray tracer with the file `balls5.nff`. Make sure that Release Mode is on (see page two). Otherwise it may take hours to render the image.



GOOD LUCK