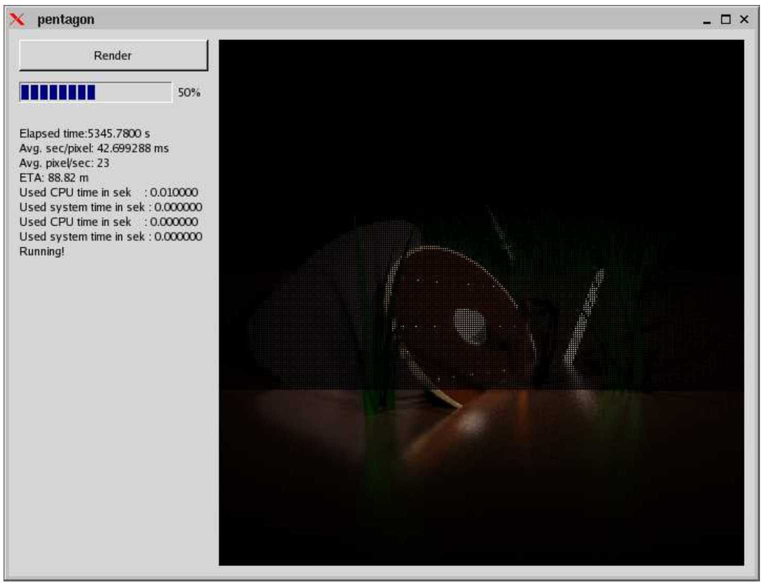
The Pentagon raytracer

ItRT E2004



```
Elapsed time:5345.7800 s
Avg. sec/pixel: 42.699288 ms
Avg. pixel/sec: 23
ETA: 88.82 m
Used CPU time in sek    : 0.010000
Used system time in sek : 0.000000
Used CPU time in sek    : 0.000000
Used system time in sek : 0.000000
Running!
```

The Penta group (group 5) is:
Jesper Fruergaard Andersen - 19960197
Bent Bisballe - 20001467
Ulrich Christensen - 20001428

Project homepage:
http://www.aasimon.org/pentagon

Group email:
pentagon@aasimon.org

15th October 2004

Contents

Chapter 1

Description

1.1 Features

Path trace features

Sampler	center, jitter, multijitter
Filter	tent
Camera	thinlens
Shapes	Triangle, Sphere, Trianglemesh
Lights	on shapes and point
Shaders	Lambertian, perfect and imperfect specular, transparent materials
Lighting	Direct lighting
Texture	Perlin noise and turbulence, stripe, mix, image

Other features

Hardcoded scenes demonstrating the featues
fileloaders : 3DS, Ply, Obj, Penta
QT GUI
Timing : Measure used time, show average speed and ETA
Multithreaded
Configuration file
Commandline parameters
Prepared to run distributed

1.2 Algorithms

1.2.1 Samplers

We have chosen to implement 3 samplers. A center sampler that simply takes the point in the middle. Jitter and multi-jitter distribution with a configurable number of samples. These are used for both the position of the ray within a pixel and the adjusting the thin-lens camera does.

1.2.2 Filter

One filter has been implemented, a tent filter. The filter can be applied zero or more times.

1.2.3 Camera

The camera adjusts the origin and direction of the rays according to the properties of the camera. The camera makes one part of the world visible. This is the field of view or zoom. Narrower field of view equals more zoom, and a smaller part of the world is visible. The thin-lens camera also makes part of the world be out of focus. At some distance from the camera everything is in focus and the further an object is away from that distance the more out of focus it is. This is achieved by simply turning the rays around the point of focus.

1.2.4 Shapes

Several shapes have been implemented. Some are shapes to be seen in the scene, others are only used internally. The shapes we have made for use in the scene are Triangle, Sphere and mesh of triangles.

1.2.5 Lights

We have chosen to only implement constant light emission throughout the surface. Light is distributed according to a phong distribution. Light is always associated with a shape. A point shape has been made to make point lights possible. Simply a fixed point and normal.

1.2.6 Shaders

Shaders are used to describe the properties of the materials on our objects. 4 basic shaders are used and a group shader. The purpose of the group shader is to allow usage of multiple basic shaders on the same object.

The diffuse shader is the only one where you can actually see the surface of the object. For perfect and imperfect specular reflection and transparent shaders the surface cannot be seen directly but can absorb some of the reflected light.

1.2.7 Texture

We have implemented several kinds of textures for use with the diffuse shader.

Perlin noice has been used to create turbulence and turbulence has been used on stripes to create different patterns. We have image mapping as a 2-dimentional texture and also ability to mix 3 different colors.

1.2.8 Direct lighting

Direct lighting is only traced on the diffuse shader. On perfect specular reflection the chance of contribution is very small and will only add noice to the image. For imperfect specular reflection the same is true if the beam of possible reflectance is narrow enough.

Shadow rays are traced to each lightsource to check for possible contribution. Weight is assigned to the lightsource according to the total light emission on a point. Since we only have constant emission that is simply the intensity of the light. One light source contribution is calculated and scaled according to its relative weight.

1.2.9 Bounding Box

To decrease rendering time the Bounding Box acceleration structure was added. This should decrease the number of checks by quickly eliminating most unnecessary checks.

How to box

Two ways of determining how to create bounding boxes have been implemented. Both of them create a binary tree of bounding boxes by recursively dividing the list of shapes into two subparts and generating bounding boxes for those. The difference between the different strategies to create bounding boxes is the way the splitting is done.

The first one is very simple and not a very intelligent strategy. The list of shapes is simply split down the middle. This approach though stupid is a considerable improvement over not having bounding boxes at all.

The second way is to maintain three sorted lists, one for each coordinate. Based on the these lists we determine the best coordinate direction to divide by. Using this we split the list (sorted by that coordinate) in two lists for which we recursively split based on the lists and their equivalents sorted by the other two coordinates. The best split is based on what split would in splitting space in two create the most even division. Returned from the splitting subroutine are two values, one is the determined coordinate to divide by and the other is the index in the list which is closest to the spatial middle. One can choose between dividing by this index or just divide the list in the middle.

On our preferred 3DS scene the second approach is slower (excluding generating the bounding boxes). On small hardcoded scenes the second one does however preform better than the simple approach.

1.2.10 Tracing process

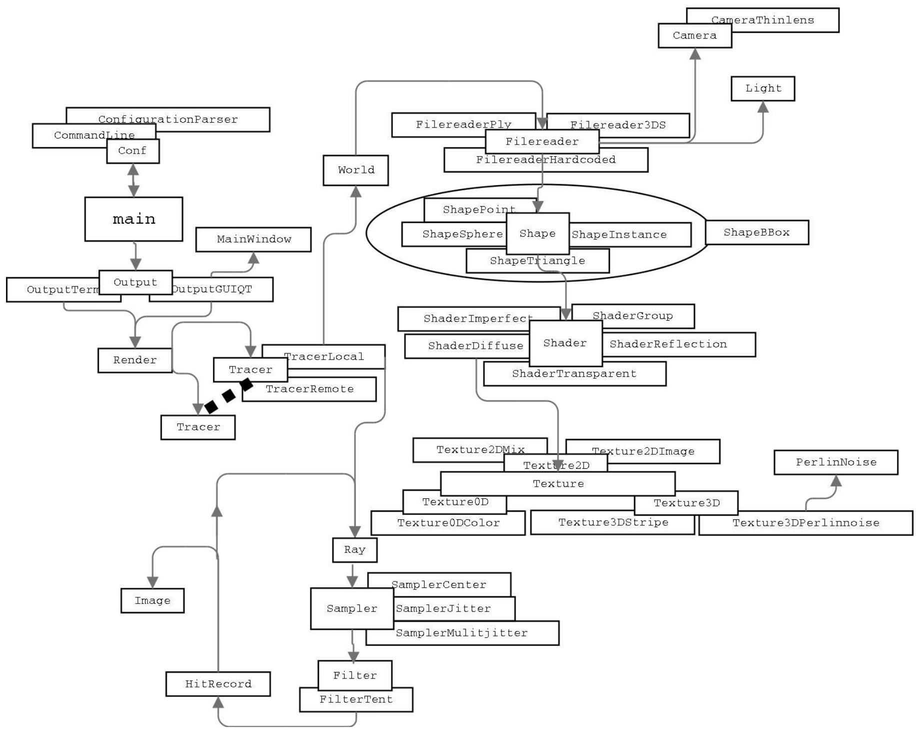
Each pixel is traced independently. The path of a fixed number of rays is traced through the scene. The average of these are used as the color of the pixel. The rays are generated according to some distribution by a sampler and filtered by applying a number of filters that moves the rays to achieve importance sampling. After this the camera is applied to for its effect.

After the ray has been adjusted it is traced through the scene. The ray is traced repeatedly and the process is stopped at each step by a fixed probability. The ray is traced to find the first object it hits. If no hit is found the ray is said to hit the background and tracing is stopped. Otherwise the shader is calculated to get a color to affect the ray and if shadow rays should be traced. Shadow rays are traced as described above.

The tracing of the ray is done in a loop by rememboring sum of how light is affected and this is applied before the lights contribution is summed.

1.3 Architectural design

Our architectural design is based upon the idea of a generic and yet solid foundation for a ray-tracer. The reason for this, is our intension of further development after the ending of this course.
Therefore we decided to implement it as generic as possible (according to the knowledge we had at that time).
Here are a sketchy overview of the entire architecture of the Pentagon ray-tracer:



As seen, the structure is quite complex, and very hard to get an overview of. Therefore, during the next couple of sections we will try to explain the parts in further details.

1.3.1 Main

Main simply parses the command-line parameters and reads the option file. The collected values are filled into the global configuration object, to be read from various places in the ray-tracer. Finally it invokes an output object, according to the parameters (currently gui or terminal).

1.3.2 Output

Initialises the output widgets, produces gui, readies the terminal, makes some coffee, and eventually creates a render object and calls it.

1.3.3 Render

The render object is a very simple object, whose part is mainly to know which pixels are to be calculated, and which have already been calculated. It starts the tracer(s) and orders them to calculate the pixels, one by one, and returns them to the output object. Other than this it does nothing.

1.3.4 Tracer

Does the actual tracing of rays. Each iteration produces one pixel. Between iterations, the tracer object sleeps.

Load scene

In order to calculate the requested pixel, the tracer has to know the scene. In order to do this, it will have to load it from disk or from a network stream, since it does not necessarily share physical memory with the renderer (It might be located on a physically disjoint machine)

Loop over generation of rays and tracing them

When the scene has been successfully loaded, and other parameters have been set (stuff not located in the scene file), the actual tracer loop starts running. This has already been discussed in a previous section.

1.3.5 Communication between Tracer and Render

The tracer runs threaded, meaning, it sleeps until it gets a new pixel request. Once the pixel has been successfully calculated, the colour needs to be returned to the render object. To do this, some shared memory has been created for it to write in (protected with a semaphore). While the calculation runs, the render object sleeps, until woken by the tracer, after writing the pixel value.
Now the render can forward the pixel to the output object, which handles the image-writing and friends, and finally, request the next pixel to be generated (if any).
As mentioned, the tracer object might be located on another machine. When this distribution is active, the TracerRemote object is put in between, as a middle-ware layer. The idea is, for the TracerRemote object, to communicate with Back-end object listening on the network, which when connected, can spawn a TracerLocal on the remote computer and do the communication transparent, just as if it were on the physically same computer. This however is one of the things we did not have time to fully implement, but the rather strange architecture surrounding Tracer is due to our preparations for this.

1.3.6 World vs. configuration

The world object contains the scene, once loaded, and informations about it when not. When not loaded, the tracer needs the configuration object, which contains the actual filename of the scene. With a correct set configuration object, the tracer will be able to load the scene, and together with the world informations about sampler, filters and which camera to use, the scene can be loaded, and the tracing initiated.

1.3.7 The back-end

The back-end object is intended to run as the listening process on the remote computer. Once a connection is requested from TracerRemote, it spawns a thread, initialised with network stream. The back-end object works much like the Render object on the local machine, sleeps until awakened. Beside spawning object on incoming network connections, the back-end does nothing.

Chapter 2

Problems

c3ds

The C3DS 'library' was our first attempt to parse and read 3ds files. This 'library' is not a real library, but rather some source-code that one can incorporate into ones project. Unfortunately the author decided not to document his work (especially not the parts that are hard to understand from the source-code), which led to a lot invalid 3ds scenes. We thought; thats OK, we'll just try to figure it out later, until we found out, that the c3ds code did not even support camera, lights or textures (it was made for reading models into a 3d real-time engine). Then we decided to use the much more complex *lib3ds*, that seemed to support everything we could be needing, and the 3ds scenes have never looked better.

Black pixels (the nan problem)

During the final stages of our ray-tracer, we spotted a lot of black pixels, which we could not understand where came from. First we tried to print their values; they said rgb(0,0,0). Then we checked the rays that produced this black colour; it said (0,0,0)... hmm.
Then we tried adding nonzero values to these zero vectors, and checked them again; they said (0,0,0)!!!
Then all math broke down. We finally came to the conclusion, that illegal operations are permitted on floats and doubles (i.e. division by zero and square-root of negative numbers), which apparently produces the value *nan* (not a number). This magic value is not affectable by any further mathematical applications (i.e. nan + 1 does **not** equal 1), which led to the black pixels. We fixed this in numerous ways, ranging from checking the values of the actual vectors and replacing them by valid ones, to checking the values of components of calculations in order to ensure their validity, prior to the execution of the calculation.

Camera position / Direction

For a long time, we had problems with the camera adjustment. The look-at point was wrongly interpreted, as being the directional vector for the camera direction. This caused all of our test 3ds scenes (except the viking3.3ds scene) to be rendered wrongly, thus forcing us to use the viking3.3ds scene for testing (Not that we wouldn't have chosen it anyway).

Bounding Box split

Due to *a lot* of bugs n the bounding box split code, we never really got to the optimisation of the strategy. But we have a functional bounding box algorithm, which splits using sorting (though not in an optimal way).

Image-maps on spheres

We had some problems getting the Image texture on the sphere to work as we wanted it to. This turned out to be an inaccuracy in our spheric coordinate mapping algorithm.

Shape instance

We had no problems what so ever implementing the ShapeInstance class prior to the bounding box implementation. **But** when the bbox was introduced, we all of a sudden had to think of a way to reverse the transformations. Due to lack of time (see next problem) we decided not to develop the ShapeInstance class further, since we could replace it by simply using ShapeTriangles added the transformation on scene load-time. Hence, no functional ShapeInstance in our final ray-tracer.

Lack of time

Our all time favourite problem, is lack of time, or work pressure if you like. Due to a lot of implementation requirements (especially during the final stages of the project) we were pressured into making half solutions, and ad hoc implementations of algorithms which we did not fully understand (due to lack of time to study them prior to implementing them). This of course leads to a lot of bugs, which could have been prevented in the first place, if we had had the time to think twice (or trice) before implementing new functionality.

Chapter 3

The run How-To

To download the Pentagon source-code, please visit our web-site at

`http://www.aasimon.org/pentagon`

The program needs `libqt v3.3.3 or later` and `lib3ds v1.2.0` in order to compile. When properly installed, all you need to do is run `make` in the source folder.
`libqt` can be downloaded from `http://www.trolltech.com` and `lib3ds` can be downloaded from `http://lib3ds.sourceforge.net/`

To compile and run from a DAIMI computer, set following system variables:

```
   declare -x QTDIR="/users/deva"
```
and
```
   declare -x LD_LIBRARY_PATH="/users/deva/lib"
```

The first one in order to compile, the last one in order to run. (NOTE: The `libqt` will only be available from this location a limited period of time)
If logged into a DAIMI computer, the full source tree can also be acquired by typing

```
   cvs -d /users/codey/.CVSHOME co Penta
```

Which will checkout the source-code to a sub-folder in the current directory, which will be called “Penta”.
Once compiled, the ray-tracer is now ready to run:

```
[deva@meru:.../Penta/]$ ./pentagon --help
Usage: ./pentagon [options]
Options:
  -w N, --width N               sets the width of the output image to 'N'
  -h N, --height N              sets the height of the output image to 'N'
  -f FILE, --file FILE          sets the input data file
  -o FILE, --output-file FILE   sets the output image file
  --sampler=SAMPLER             where SAMPLER is either
                                'center', 'jitter' or 'multijitter'
  --samples=N                   sets the number of samples for the sampler.
  --conf=FILE                   sets a configuration file to be used
  --novisual                    makes the program run without a GUI
                                (good for running big render-jobs)
  --visual                      makes the program run with a GUI (default)
  -v, --version                 output version information and exit
  -?, --help                    display this help and exit
```

In the cvs tree, we have put a number of 3ds scene-files. To render the scene used throughout this assignment and on the web-page, type

```
   [deva@meru:.../Penta/]$ ./pentagon -f 3ds/viking3.3ds
```

The raytracer defaults to 16 rays per pixel using the multijitter sampler. To override this use the `--sampler` and `--samples` parameters.

Group email:
`pentagon@aasimon.org`