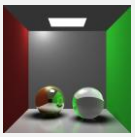


2014/2015, 4th quarter

INFOGR: Graphics

Practical Assignment 3: Advanced Shader Programming

The assignment:

For this third practical assignment, you will do some more advanced shader programming. The procedure is a bit different than before. This assignment will feature various tasks of different levels of difficulty: *Easy*, *Medium* and *Hard*. There are 6 easy, 6 medium and 5 hard tasks. You will be able to compose your assignment by choosing from these tasks (cf. *Grading*). As preliminary knowledge we expect you to have completed the two previous assignments, and understand the workings and purpose of the things you implemented.

> ***Easy tasks:*** The style of these tasks is similar to the assignments in practical 2. You will get background information, an algorithm description, formulas, and hints. Yet, most of the tasks are more difficult than the ones in practical assignment 2.
>
> ***Medium tasks:*** For the medium tasks you are required to do more research on your own. You will get some background information and a high level overview of the algorithm, but there are much less hints and no code fragments. Also the medium tasks generally require more complex implementations than the easy tasks. The high-level algorithm overview contains no specifics for the code, so optimizations are entirely up to you and can improve your grade.
>
> ***Hard tasks:*** The hard tasks are meant for excellent students who are willing to put more effort into the practicals than normally expected. These tasks are more difficult than the others, but also offer more freedom in solving them. You are only presented with an advanced topic and some references, but the research and implementation are your own responsibility. Doing one of these assignments is highly encouraged – but as can be seen under *Grading* – not mandatory.

Starting point:
You can use your end result for assignment 2 as a starting point for this assignment. If you didn't do so well on your implementation for assignment 2, but can show us that you are very motivated to do a good job for assignment 3, the teaching assistants can help you implement everything that is needed to get you started. Furthermore, you are encouraged to use Shader Model 3.0, mostly because of the larger number of arithmetic operations that have to be performed in the shader. Finally, please create a new solution for this assignment called "GraphicsPractical3" and *copy over* any code that you need from the previous assignment. Make sure to use the correct *namespace* and please *remove all unused code* from your implementation.

Reference material:
The HLSL reference guide [1] in the *Microsoft MSDN library* gives very useful information about data types and built in functions and provides several examples for good HLSL programming. References about the theoretical background are listed throughout the assignment. But notice that we also expect you to do your own research if you feel that this is necessary for you to solve a particular task.

Some references have C#/XNA code samples that use XNA Game Studio 3.1. Since we use the latest version XNA Game Studio 4.0, it is possible that there are some differences and that not all examples work "out of the box". You are expected to deal with this yourself. A very handy resource for this problem is the list of major changes in XNA 4.0 [2] by Shawn Hargreaves.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/bb509638(v=vs.85).aspx
[2] http://blogs.msdn.com/b/shawnhar/archive/2010/03/16/breaking-changes-in-xna-game-studio-4-0.aspx

Academic honesty:
An implementation that is copied 1:1 from a reference implementation or tutorial is considered plagiarism and will not be graded. However, in order to expand your knowledge about the subject, you are of course allowed to look for and read tutorials, reference implementations, and other resources. For some tasks, it is even expected and necessary to do so. You should just not copy any of the provided code, but use the resources to get a better understanding of the theory and reasoning behind the concept, and use that knowledge to write your own implementation. Naturally, your code will sometimes look quite similar to the one used in a resource that you found.

But if you re-implement it yourself, adapt it to your coding style and program structure, proved useful comments, optimize it, etc. it should be easy for you to demonstrate that you really understood the underlying principles and did not just copy-paste and change some variable names. In any case, you *have* to explicitly document in the read-me file exactly which resources you used, and you *must* write complete and good commenting for every non-trivial piece of code. If you do not, and it is obvious to us that you used a reference implementation, you could end up getting 0 points for the assignment due to plagiarism.

Commenting:
Do not underestimate commenting! It will probably be about as much work as the implementation itself.

Grading:
Your assignment can be composed as follows:

- You are *required* to do **4 out of 6 easy assignments** (except when opting for H6), and by doing this you can get a maximum grade of **8** (up to 2.0 per assignment). You **cannot achieve more than 8.0 points with "easy" assignments**; if you work on five or six, the four best solutions will determine your score.
- You can earn 2 more points to get a maximum grade of **10** by doing *either* of the following:
  - **2 medium** assignments.
  - **1 hard** assignment (H1..H5).
- Alternatively, you can opt for the H6 assignment (ray tracing), in which case your grade is based purely on this assignment; see the assignment description for details.

It is not forbidden, but we do not recommend submitting more than the required medium or hard assignments, because it is generally better to do few things very well rather than having many mediocre solutions. If you decide to do so nevertheless, we will grade what we consider to be the best one.

The following rules from the previous assignments still apply:

- Your code has to compile and run on the machines in BBL 175, so if you work on other computers make sure to do a quick check there before you submit it. If this requirement isn't met, your work cannot be graded and your grade will default to 0.
- Please *clean* your solution before submitting (i.e. remove all the compiled files), see *Appendix 1A* of the Tutorial for more information. After this you put the contents of the solution directory in the zip file.
- When grading, we want to get the impression that you really understand what is happening in your code, so your source files should also contain comments to explain it. (Besides, every good program should be properly documented!).
- Finally, we also want to see a consistent and well readable coding style. Use indentation to indicate structure in the code for example. Don't worry about this too much; if it is readable and consistent throughout the whole project, you should be fine.

End result:
Like the previous assignments, you have to hand in your complete end result. This means all your source code, project files, content files and a read-me (cf. *Deliverables*). You no longer have to send in screenshots of intermediate results, but we require you to show us what you did in another way (see the *Setting up* part of the assignment).

If you did something that looks "nice and cool" you are however strongly encouraged to send us a screenshot of it or a link to a YouTube or Vimeo video that demonstrates your implementation. We will collect the nicest results and put them in a ***hall of fame*** on the course's website, use them for advertisement of our study program, and preserve them for admiration by future generation of students:

Hall of Fame 2010-2011: https://vimeo.com/42925649
Hall of Fame 2011-2012: http://www.cs.uu.nl/docs/vakken/gr/2011/gr_fame.html
Hall of Fame 2012-2013: http://www.cs.uu.nl/docs/vakken/gr/2012-13/gr_fame.html

Notice that some of the reference images in this assignment are solutions from students who followed the course in previous years. Please put your screenshots in a separate directory and document them along with potential links to your video(s) in the read-me file.

Deliverables:
A ZIP-file containing:
(a) **The contents of your (cleaned) solution directory**
(b) **The read-me (in the .txt file format)**

The contents of your solution directory should contain:
(a) Your **solution file** (GraphicsPractical3.sln)
(b) All your **source code**
(c) All your **project files** and **content files**

The read-me file should contain:
(a) **The names and student IDs of your team.**
2-3 students; penalties for submitting with less or more team members will apply
(b) **A short statement specifying how you split the work among the team members.**
Shortly explain how you worked together and/or split the work. No long explanation needed, just a short description, such as "we regularly met and implemented everything together" or "Paul implemented A and B, Tom did C and D, and the rest we did together" or whatever you did. Obviously, we hope not to find any statements such as "Paul and Tom implemented everything and Wolfgang just put his name on the delivery". Be aware that we will randomly select a few groups for a short verbal interview in order to verify that your statements are correct.
(c) **Short (!) comments on your code related to the mandatory assignments.**
Most importantly: Internet link(s) to any reference implementation that you have used. Even if it is an implementation that is listed in the assignment's provided references. If we find out you used one that isn't listed in the read-me file, we could consider it to be plagiarism (cf. *Academic honesty*).
(d) **A statement about which assignments you have implemented and related information that is needed to grade them, including detailed information on your implementation.**
We will not make any wild guesses about what you might have implemented nor will we spend long times searching for special features in your code. If we cannot find and understand them easily, they will not be graded, so make sure your description and/or comments are clear.

The **contents of the zip file** should look like this:
/GraphicsPractical3.sln
/GraphicsPractical3
/GraphicsPractical3
/GraphicsPractical3Content
/readme.txt
That is, put the contents of your solution directory and the read-me file (in the .txt file format) in the **root** of the zip file.

Note that any violation to these rules will have negative consequences for your grading. Also notice that the read-me file should be well readable. It is part of the program that you are producing, so the rules about "consistent and well readable coding style" apply to it as well.

Mode of submission:
- Upload your zip file before the deadline via the SUBMIT system at http://www.cs.uu.nl/docs/submit/
- Make sure to upload it to the correct entry, i.e. not the ones for late delivery if you are submitting on time
- Note that we only grade the latest submitted version of your assignment, so if you upload to the late delivery entries your earlier submission will be discarded.

Deadline: Tuesday, **June 30, 2015, 23:59**

If you miss this deadline, there will be a second entry in the submit system to upload your solutions. It is open 12h longer, i.e. till **July 1, 2015, 12:00**. Uploading to this entry will result in a deduction of 0.5 in your grading.

If you miss the second deadline as well, there will be a third entry in the submit system. It is open 24h longer, i.e. till **July 1, 2015, 23:59**. Uploading to this entry will result in a deduction of 1.0 in your grading.

If you the miss third deadline as well, contact the instructor of the course as soon as possible. Based on your reasons for the delay, he will decide if you get further reductions or if your assignment will be graded with a 0.

If you don't miss any deadlines, but upload your submission to the wrong entry in submit, you will get the same deduction as if you had submitted it too late, so be careful.

Good luck and have fun with the assignment!

Setting up

For this assignment, you don't have to take screenshots of the intermediate results anymore! Rather than that, you should implement a **demo camera** reminiscent of the Unreal Engine Features Demo [1]. Basically we just want a camera position for each implementation that clearly shows what you implemented. We want to be able to **cycle through your implementations** of the various tasks by pressing the **space bar**.

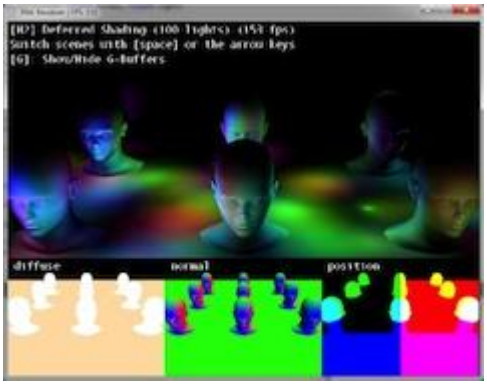
We also want to be able to see the **robustness** of your implementation. This can be as simple as allowing the model to rotate or the light source to move etc. so we can see how your implementation holds up in different situations. And we want some ***information*** to be displayed about what we are seeing. At the very minimum, the name of the task you implemented. This can be achieved for example by using the `SpriteBatch` object to draw some text in the window.

You will also be supplied with a new **model** to use. You can find it on the Graphics course website. To get it properly visible in your window, assuming the screenshot camera settings from the previous assignment, you can use a scaling of 0.5.

References:

[1] http://www.youtube.com/watch?v=7v2ExTxlf7c

Content Overview



Easy Assignments (choose four):

E.1 Multiple Light Sources
E.2 Spotlight
E.3 Cell Shading
E.4 Frustum Culling
E.5 Color Filter
E.6 Gaussian Blur

Medium Assignments (choose up to two, or up to one hard one):

M.1 Textured Light
M.2 Shadow Mapping
M.3 Reflection
M.4 Cube Mapping
M.5 Transparency
M.6 Bloom
M.7 God Rays (Volumetric Lighting)

Hard Assignments (choose up to one, or two medium ones):

H.1 HDR Lighting and Tone Mapping
H.2 Deferred Shading
H.3 Parallax Mapping
H.4 Screen-space Ambient Occlusion
H.5 Image-based lighting (see M.4 for description; H.5 is the “hard” variant of M.4)
H.6 Ray Tracing

EASY TASKS

[E1] Multiple light sources

Category: *lighting*

So far, you have always used only one light source at a time. For this assignment, you should use *five lights* in your scene. You have to make this system *scalable*, meaning that you cannot use top-level variables like `Light1`, `Light2` … `Light5` in your shader. It is however allowed to define the number of lights in your shader. For instance, you can write `#define MAX_LIGHTS 5` in you shader code. Changing this number, e.g. from `5` to `10` (should we decide to use 10 light sources), should work as expected in your system. Instead of calculating the light contribution for only one light, you now calculate it for all lights and accumulate the result to get the total light contribution.

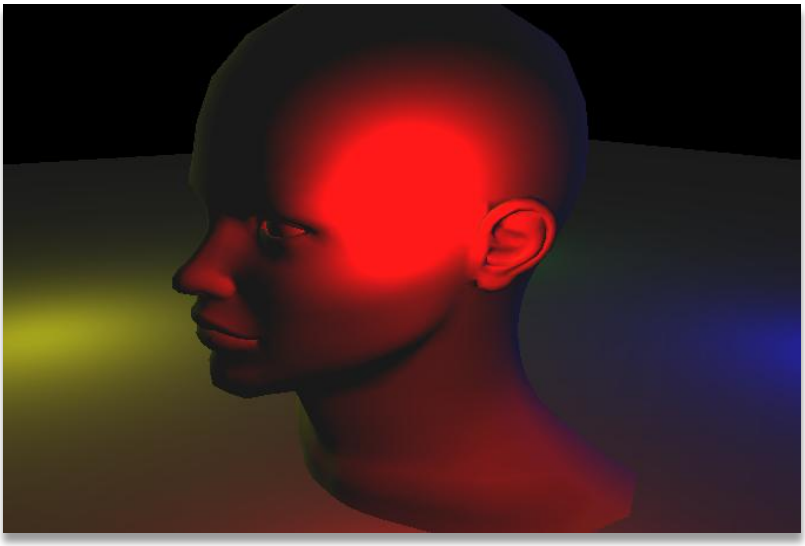
Exercise E1: Add support for *multiple light sources*. Extra credit can be earned by also creating *colored lights*!

Note 1: The `SetValue` method of the effect's parameters is overloaded to also let you pass arrays of any compatible data type (e.g. `Vector2[]`).

Note 2: HLSL also supports arrays (e.g. a top-level variable: `float3 LightPositions[5]` represents an array that can hold five 3-dimensional vectors).

Note 3: `#define` is a *pre-processor directive*. It replaces every occurrence of the first argument (here `MAX_LIGHTS`) with the second argument (here `5`) before the shader is compiled.

Reference image:



[E2] Spotlight

Category: *lighting*

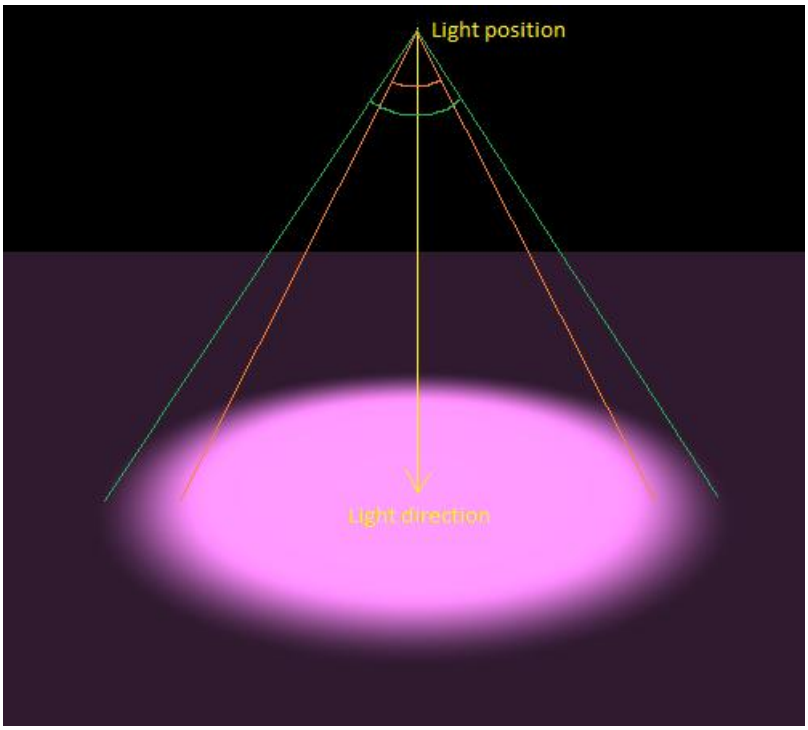
Earlier, you encountered *directional lights***,** which shine an infinitely wide bundle of light in one direction. Also you have seen *point lights*, light sources that emit light from an infinitely small point in space in all directions. Using these, you can simulate a variety of different light sources. But which one would you use if you wanted to simulate, say, a flashlight? The light that comes out of a flashlight has a point from which the light is emanating *and* it has a direction. Also, its bundle is not infinitely wide, but shaped like a cone. This is where the *spotlight* comes in.

A spotlight has a position, a direction and *two angles* to represent the cone-shaped light bundle. These two angles are called the *inner* and *outer angle*. Take a look at the image below. The parts of the purple surface that are in the inner cone (orange) of the light, are lit as they would be by a *point light* (not a directional light!). The parts of the surface that are in the outer cone (i.e. between the green and orange lines) receive linearly less light as they are positioned closer to the edge of the outer cone. Parts of the surface that are outside the outer cone are not lit by the spotlight at all. The orange and green *arcs* in the reference image represent the inner and outer angle respectively.

Exercise E2: Implement a *spotlight*.

Note 1: A spotlight is much like a point light, but with more restrictions on the light intensities.
Note 2: You do not have to take *distance attenuation* into account.
Note 3: In case you are planning to do assignment *[M1] Textured Light* and/or *[M2] Shadow Mapping*, notice that it depends on this one.

Reference image:



[E3] Cell shading

Category: *nice shader hacks*

A technique that has become popular over the last years is cell shading. Modern games like *Borderlands* and some of the modern franchise of *The Legend of Zelda* series are examples if this style.

Cell shading features large monochrome cells on the object. These are caused by using only a very limited number of shades of each color. Usually the result of our shading looks very smooth and continuous. For cell shading however, we split the color range into a few large intervals, similar to the image below.

After calculating the lighting of a point as usual, we map the value to one of these intervals and give use that color for the model. This results in hard borders and a very cartoony graphics style, while still retaining the illusion of 3D due to the lighting.

Exercise E4: Implement cell shading with smooth (non-aliased) cell borders. Note that while cell shading can be performed as post-processing process, we want to see an implementation directly in the pixel shader.

Note 1: Just discretizing the colors will result in aliasing around the border of the cell. A principled solution to this problem is to smoothly blend between colors in the boundary regions using the “DDX”, “DDY” commands to estimate how quickly the color will change towards adjacent pixels. Alternatively, you can also use texture lookups with interpolation for (some, non-adaptive) smoothing, which will not give you full points for the exercise. For full score, address this problem and explain your solution briefly in your readme file (a perfect solution is not required).

[E4] Frustum culling

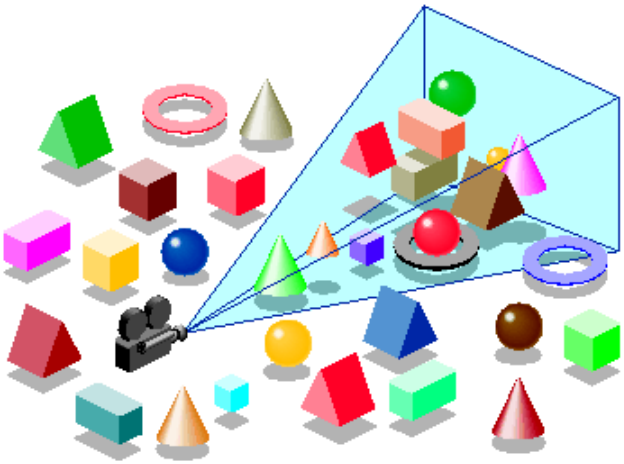
Category: *performance*

So far, you have always used a small amount of models in your scene. If you use more models, more and more triangles get passed to the graphics card, including triangles of objects your camera does not even see (e.g. objects that are completely behind the camera). More formally, the object's *bounding volume* (e.g. *bounding sphere* or *bounding box*) does not intersect the camera's *view frustum*. XNA has functions to create a bounding volume for your models and for your view frustum. It can also check whether two volumes *intersect*. The earlier you can remove triangles from the rendering process, the better it is for your application's performance.

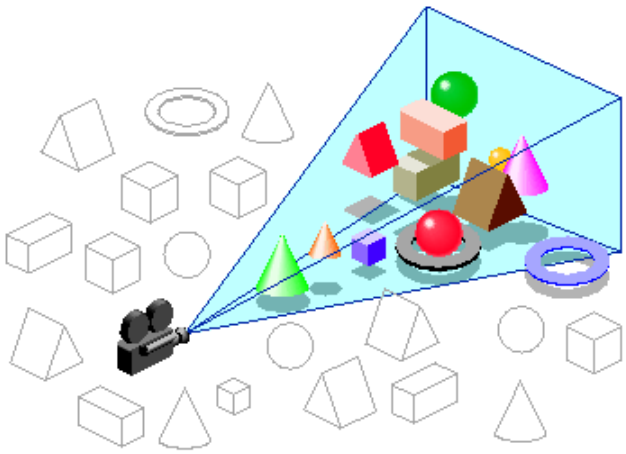
Exercise E4: Implement frustum culling by testing for each model in your scene whether it is visible to the camera. If it is not, you remove this model from the rendering process (i.e. you do not draw this model). Please also implement a *counter* that displays how many models are drawn and how many are culled.

Note 1: You don't have to check intersections for the quad you created in assignment 2, if you are still using it. But if you want to do this, by all means do.
Note 2: To show us your algorithm, you may want to draw some more models!
Note 3: The availability of XNA's bounding volume functions probably makes this the easiest task.

Reference images:



Without view frustum culling



With view frustum culling

Remark (not relevant for grading): View frustum culling is one of several approaches for *output-sensitive rendering*, i.e., for achieving rendering times that do not depend strongly on scene complexity but rather on resolution and other measures of visual fidelity. Other methods include multi-resolution representations (reducing the number of triangles for far away objects), and occlusion culling (not rendering invisible objects). All three of these aspects need to be brought together for strong output-sensitivity (i.e., for being able to render extremely large, open worlds, as in many modern games). Obviously, a full solution is a bit beyond the scope of this assignment…

[E5] Simple color filter

Category: *post-processing*

For this assignment you will implement a post-processing filter that takes the resulting image from rendering the whole scene, and turns it into a *gray-scale* image. See the images below for an example.

What exactly is post-processing and how can we do it? You can let your program render your 3D scene to a texture instead of to the back-buffer by changing the `RenderTarget`. Manipulating this texture with pixel shader(s) before displaying it on the screen to make the end result look different is called *post-processing*. A post-processing effect is applied to the content of the frame-buffer after rendering, not to the 3D data in world-space. For high speed, it is desirable to perform post-processing using pixel shaders, and this should be done in this practical. The manipulation of the texture with pixel shader(s) can be done in two ways:

1. You can draw a quad that covers the whole screen (i.e. a full-screen quad), using the result of your render as a texture. You can then sample every pixel using the texture sampler, as you did for the texturing assignment in the previous practical. This is the traditional way to do post-processing when using DirectX or OpenGL.
2. The `SpriteBatch` object has functionality that lets you draw a (full-screen) image and apply a pixel shader to it. This is a more specialized and easy to use solution, but it is specific to XNA.

Finally you apply the gray-scale function to every pixel. This function is a fixed weighted sum of the color channels: $Y = 0.3R + 0.59G + 0.11B$. These weights are determined by relative sensitivity of our eyes to certain colors. They are most sensitive to green, hence the higher weight for green, since we perceive this color to be brighter. Naively averaging the colors with equal weights produces an image that has an unnatural brightness (the difference might be subtle in practice, though; nonetheless – we want this here!).

Exercise E5: Implement a *gray-scale post-processing filter* using the color weights described above.

Reference images:

Original image

Grayscale image

References:

[1] http://en.wikipedia.org/wiki/Video_post-processing (The “Uses in 3D rendering” part)

[E6] Gaussian blur

Category: *post-processing*

Blurring a 2D image is useful for a variety of purposes, and is used not only in computer graphics, but also in image processing and other fields. Blurring images for example reduces noise, of course, at the loss of spatial resolution (your cell-phone camera can tell you more about that one; in particular, in low light…). Mathematically, blurring corresponds to averaging the color of a pixel with itself and the colors of surrounding pixels. The simplest blurring techniques is uniform averaging of the pixels in a square area around the target pixel (i.e., with equal weights); this is called a *box*-filter (if you think of the weight function as a height field, it looks like a box on a plane).

Another way of thinking about this is to define a *convolution kernel*, which is a function of weights (this is just a matrix of weights; a box function for the simple case above; see figure on the right); this "kernel" is then moved over the image and the weights are used to determine weighted averages, written to the pixel where the kernel is centered over (cf. [3] for an example). Convolution kernels should usually have an odd width and height so that they have a clearly defined center. Also, usually the sum of the weights in the kernel is equal to 1. This ensures the overall "energy" of the image is preserved, i.e. there is no overall loss in intensity, in other words, it computes a weighted average (rather than a general linear combination).

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Simple averaging

When using a box blur, the box shape might still be slightly visible and produce artifacts in the blurred image[1]. This can be prevented using a circular kernel. The most commonly used algorithm for blurring a 2D image is the *Gaussian blur*. Instead of using the same weights for every element in the kernel, it uses values produced by the *normal distribution* (Dutch: *normale verdeling*). You can find a pre-generated 7x7 Gaussian convolution kernel on [1].

The process of implementing a Gaussian blur filter for your image thus consists of two steps.

1. Implement convolution with 2D convolution kernels.
2. Use a pre-defined Gaussian convolution kernel to average the pixel colors.

The Gaussian kernel is *separable* (cf. [5]). Because of this, the algorithm can be (heavily) optimized by splitting the convolution in a horizontal pass and a vertical pass.

Exercise E6: Implement a *Gaussian blur post-processing filter*. You have to optimize the kernel by splitting it in a horizontal and vertical kernel to get the full points for this exercise.

Reference image:

Original image *Gaussian blur*

References:

[1] http://en.wikipedia.org/wiki/Gaussian_blur
[2] http://elynxsdk.free.fr/ext-docs/Blur/Fast_box_blur.pdf (Very good explanation)
[3] http://www.songho.ca/dsp/convolution/convolution2d_example.html
[4] http://www.dhpoware.com/demos/xnaGaussianBlur.html
[5] http://blogs.mathworks.com/steve/2006/10/04/separable-convolution/

[1] A full explanation is given by sampling theory, which states that the box shape creates aliasing frequencies in the result; but this goes too far for this assignment. If you are curious, check this out: http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem, or look at chapter 9 in the text book.

MEDIUM TASKS

[M1] Textured light

Category: *lighting*

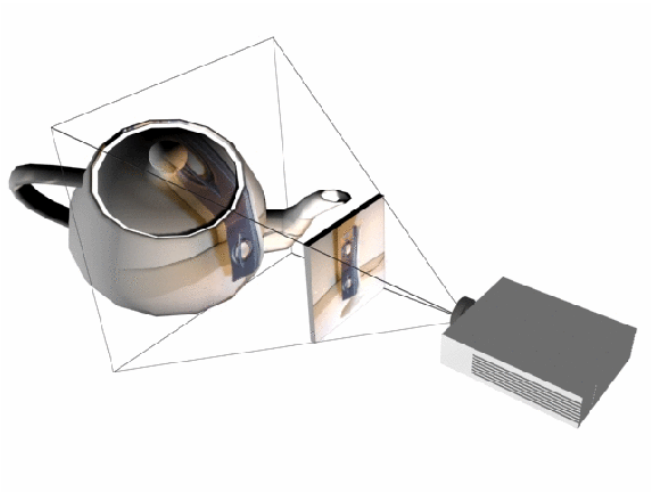
Let's add another bit of complexity to the *spotlight*. At present, the spotlight color is just a simple single color. We could also sample the color from a texture, to create a slide projector or beamer effect. It is the same as the projective texturing in the reference below.

High-level algorithm description:
Instead of using a constant light color for the spotlight, you sample a texture. As with shadow mapping, you render the scene as usual, but you also transform *every vertex with the matrices of the light camera*. If a pixel falls into the light's view, you remap its 2D position (in the [0, 1] region) to a texture coordinate (in the [-1, 1] region) and you sample the texture for the correct color.

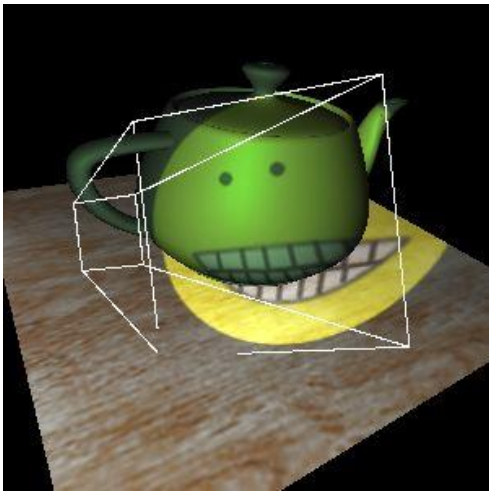
Exercise M1: Implement textured lighting (also known as “projective texturing”) for one light source.

Note: Because this task can be implemented in a very simple way, you will be judged extra on the commenting, quality, robustness and utilized optimizations for your implementation.

Reference images:



Schematic of textured lighting



Result of textured lighting

References:
[1] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Projective_texturing.php

[M2] Shadow mapping

Category: *lighting*

Simulating realistic shadows is a problem that is almost as old as 3D computer graphics itself. It has been around for so long, that it might seem like a problem that should be almost trivial by now, but it turns out it is not! Creating realistic shadows in games is still a very hot topic.

For this assignment, you will add another level of realism to your scene. You will implement a technique called *Shadow Mapping*, nowadays probably the most widely used technique to simulate shadows.

High-level algorithm description:
If you look at the scene from the *point of view of the light source*, you essentially see all the pixels that receive light from it. The only actual thing you are interested in is the *depth* of the pixels you see (i.e. the distance of the pixels to the light source).

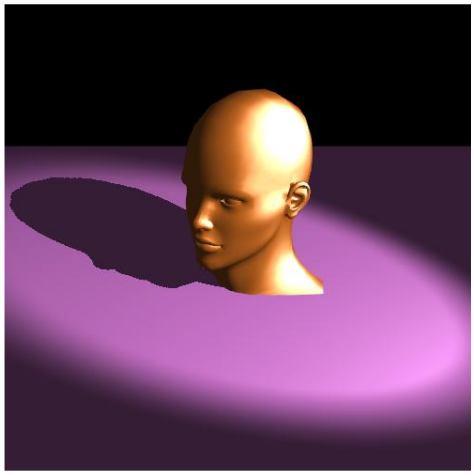
1. Render the scene from the point of view of the light, store the depth (*z-value*) of every pixel as a color in a texture. These values lie between the depth of the *near plane* and the depth of the *far plane* (both of which you specified the depth in the camera's projection matrix), so they may have to be scaled to the [0, 1] region.
2. The result from this render is a texture image called a *shadow map*.
3. Render the scene as usual, but you also transform *every vertex with the matrices of the light camera*.
4. If a pixel can be seen by the light source, it is lit by the light source. This means the pixel has to fall into the light's view and its depth has to be *equal* to that of the pixel in the shadow map. If the depth is *larger* than in the shadow map, the light source does not see the pixel because there are other pixels blocking its view of the pixel, and thus it is in shadow.

Exercise M2: Implement Shadow Mapping for at least one light source. If you did exercise E1, implementing shadow mapping for all light sources is a plus, but not mandatory.

Note: For this assignment you must use the Shadow Mapping technique in conjunction with your advanced lighting model!

Hint: When you use a *spotlight*, you basically have all the information you need to look from that light to the scene. Shadow mapping for a point light or a directional light is also possible, but it is significantly harder to map the visible pixels to a (finite) 2D render target.

Reference image:



References:
[1] http://en.wikipedia.org/wiki/Shadow_mapping
[2] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Shadow_map.php
[3] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Render_to_texture.php
[4] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Projective_texturing.php
[5] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Real_shadow.php

[M3] Reflection

Category: *miscellaneous*

For this assignment, you have to simulate perfect light reflection on a flat surface, or in other words, simulate a flat mirror. The mirror is just a quad, two connected triangles that lie in the same plane.

High-level algorithm description:

1. Mirror the camera with respect to the mirror's plane (the position and the viewing direction), thus looking from behind the mirror into the world.
2. Render the scene as viewed by the mirror camera, and store the result in a texture.
3. Render the scene normally, and apply the correct part of this texture to the mirror's surface, i.e. only the part that is seen through the mirror. You find this part by transforming the mirror's vertices (i.e. the corner points) with the mirror camera's view and projection matrices.

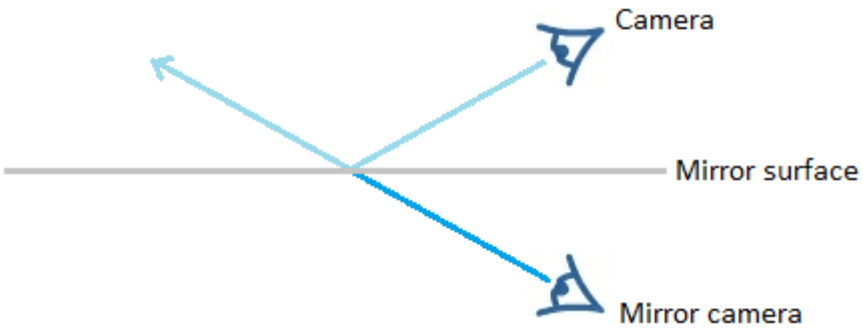


Illustration of mirror camera's position and viewing direction

Exercise M3: Implement reflection for a quad that acts as a mirror.

Note: Looking at the reference image, after mirroring the camera with respect to the mirror's plane, you will probably just see the back of the wall that the mirror is attached to. More generally, you would possibly see any geometry in between the mirror and the mirror camera. You should make a *user defined clipping plane* to remove all that geometry first, if you have any.
Hint: Rendering the whole scene from the perspective of the mirror camera, only to apply a small part of that render to the mirror's surface is a bit wasteful. The *stencil buffer* can help select the pixels of the mirror that have to be rendered from the mirror camera's point of view (cf. [4] and [5] for an example). Using the stencil buffer can also remove the need for a separate render target for the mirror camera.

Reference image:

Mirror in Max Payne 2

References:

[1] http://en.wikipedia.org/wiki/Reflection_(computer_graphics)
[2] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series3/Render_to_texture.php
[3] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series4/Reflection_map.php
[4] http://en.wikipedia.org/wiki/Stencil_buffer
[5] http://iloveshaders.blogspot.com/2011/05/using-stencil-buffer-rendering-stencil.html

[M4] Cube mapping

Category: *lighting (non-local methods)*

Simulating reflections is difficult using rasterization methods (with recursive raytracing, it is straightforward, but that also comes at a cost). In assignment M3, we have seen one way to cover a special case: A single planar mirror can be simulated using the stencil buffer and two rendering passes with a reflected camera in the second.

We now look at an alternative approach, known as "reflection mapping" or "environment mapping" [1]: We now permit an arbitrarily shape reflector, but the assumption is that the environment is far away. (So far, that only the direction of the reflected ray matters, not the position where it was reflected of the surface. Think about this – for reflection of sky & landscape in a small object, this is a valid assumption. For looking at yourself in a mirror, this is less of a useful model.)

The algorithm works as follows: The environment is stored in an omnispherical *environment map*, that specifies the appearance in all possible directions. In practice, this is usually done using *cube maps* [2], which use six faces of a cube to store an image of the environment. In the pixel shader, a special instruction exists (texCUBE) that looks up the pixel color by *direction* of a vector (it takes a 3D vector and a cube map as input and returns a color).

Description:
Create reflective environment mapping in the following steps:

- Setup a cube map of the environment (one example cube map is provided with this assignment sheet, taken from [3], where you can find many more).
- Implement a suitable vertex / pixel shader pair that computes the reflected view ray (as reflected of the surface) for each fragment on the surface of the object.
- Use a texCUBE lookup to retrieve the environment color.

Hints:

- The most difficult part is often to get the cube's side textures imported correctly and mapped to the faces of the cube map without wrong rotations / reflections. This might involve some trial and error (we will check that there are no seems in your solution, so be careful!).
- Using the right local coordinate frames is important for computing the correct reflections.
- For debugging / getting started, we recommend to start with a normal-direction texture lookup, and then add the reflection (build incrementally!).

Exercise M4: Implement a partially reflective object using a diffuse and specular cube map (image b below).

More information (optional; if you incorporate this, you can get more points, see below):

- Speaking of normal direction lookups: Aside from debugging, this is useful for so called "image based lighting" [4]. By pre-convolving the cubemap with the positive-cosine function of Lambertian reflection, one can create a precomputed diffuse light-map that is fully consistent with the reflection map and represents diffuse illumination from all directions simultaneously (although, without shadowing). By adding a second such diffuse cube map, and using a normal-direction texture lookup, you can get a nice image (this part is purely optional).
- One more useful trick: By attenuating reflections that bump of the surface near perpendicularly, you can get the appearance of a non-ideal reflector (such as glass or water). See http://en.wikipedia.org/wiki/Schlick's_approximation for an explanation.

"Hard" version:

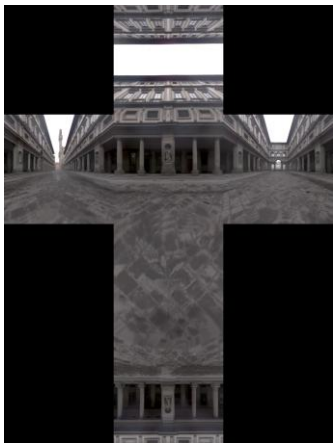
Exercise H5: Implement all the optional parts (diffuse IBL, approximate Fresnel term, compositing), too, to obtain 2.0 bonus points.

(a) bunny diffuse
(IBL, for “hard version” H5)

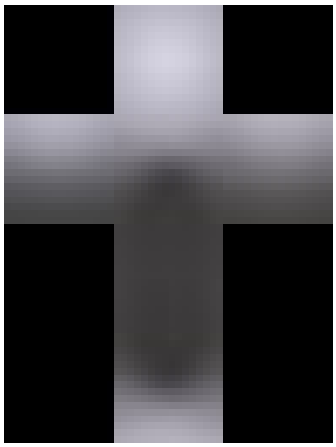
(b) bunny reflective
(environment mapping; this is your task in M4!)

(c) stronger reflections at shallow reflection angles using a $\cos^k \theta$-term
(optional for M4, required for H5)

(d) combining (adding) (a) and (c)
(“hard” version H5)



(e) Environment map (unfolded cube) used for reflections (taken from [3], copyrigh Paul Debevec).
Provided as additional matrial.



(f) Cosine-blurred version, to be used as a diffuse light map (for H5).
Provided as additional matrial.

References:

[1] http://en.wikipedia.org/wiki/Reflection_mapping
[2] http://en.wikipedia.org/wiki/Cube_mapping
[3] http://www.pauldebevec.com/Probes/
[4] http://www.pauldebevec.com/Research/IBL/

[M5] Transparency

Category: *miscellaneous*

Some objects in the real world are partially transparent, i.e. they let some light shine through them. Examples are colored glass, plastics, wings of insects etc. In computer graphics, it is often desirable to simulate these semi-transparent surfaces. However, just taking the transparency value (or *alpha*) into account when rendering transparent surfaces is not enough. So far, the Z-Buffer assumed that no pixels are transparent, and thus when evaluating whether a pixel should be drawn, it would just check if it's depth was smaller than the one currently in the same place in the buffer. But now, if a pixel is transparent, it should *blend* the color of the two pixels, rather than completely draw over the old one. This is called *alpha blending*.

Another important consideration is the *drawing order*. If you for instance want to draw two overlapping surfaces, the one closest to the camera is fully *opaque* (i.e. not transparent), and the one behind it (w.r.t. The camera) is transparent. You would expect the opaque surface to occlude the transparent surface behind it. But, if the opaque one is drawn first, and the transparent one is drawn second, the resulting image would contain a blend of the opaque and transparent surfaces colors, which is not what you want. The key to solving this problem is to draw all your geometry *back-to-front*, i.e. draw geometry (models, triangles, pixels) with a greater distance w.r.t. the camera before drawing geometry with a smaller distance.

High-level algorithm description:

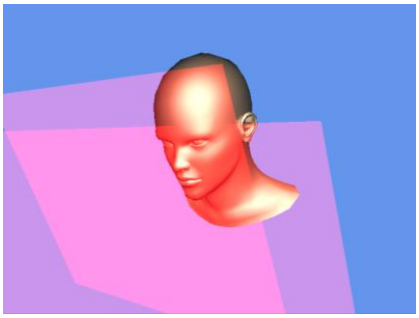
1. Sort the geometry in the scene in back-to-front order.
2. Draw the geometry in back-to-front order, using *alpha blending* to blend overlapping semi-transparent pixels.

Exercise M5: Implement *transparency* for a couple of *quads* (i.e. more than one, so we can see how the blending is done for overlapping semi-transparent surfaces). Your implementation is judged on robustness, i.e. how it holds up in different situations (e.g. if blending is still correct if the quad is positioned such it partially occludes the opaque geometry and/or other quads). To be able to see this clearly, please use different colors for each quad.

Note 1: Several different *blend states* exist for alpha blending, that describe how the colors should be blended.
Note 2: For this assignment, it will suffice to implement *transparent quads*. This makes the depth sorting much easier. If you want to implement generic transparency (i.e. for any kind of 3D model), a more complicated technique has to be used like *order independent transparency*. You are encouraged to try (it is definitely not impossible), but the simpler case of transparent quads will suffice.

Hint: For opaque geometry, it is more efficient to render front-to-back. Back-to-front is actually the most inefficient way to draw opaque geometry, because all the geometry is fully drawn, even if it is later overlapped by other geometry that is closer to the camera. When drawing front-to-back, only pixels are rendered that have a smaller z-value than the pixels already in the z-buffer.
So as an optimization, first the opaque geometry is drawn front-to-back, and after that the semi-transparent geometry is drawn back-to-front. It is also more efficient because when treating opaque geometry as “semi-transparent geometry with an alpha value of 1”, blending is still performed, which comes at a cost. When drawing the opaque geometry in a separate pass, you can disable alpha blending completely.

Reference image:
(please use a different color for each quad; the reference image shows only a monochrome solution. In such situations, the ordering of the individual quads would not matter).



References:
[1] http://msdn.microsoft.com/en-us/library/bb173417(v=vs.85).aspx
[2] http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2/Alpha_blending.php
[3] http://cg.informatik.uni-freiburg.de/course_notes/graphics_08_transparencyReflection.pdf
[4] http://en.wikipedia.org/wiki/Binary_space_partitioning

[M6] Bloom

Category: *post-processing*

A very nice post-processing effect is the so-called *bloom* effect. It is an effect that creates a glow around bright areas in an image. The effect consists of multiple rendering passes that you have to combine to get the final result.

High-level algorithm description:

1. Store the result of a normal render of the scene into a texture.
2. Apply a *bright pass* to the image, i.e. remove all pixels with a *luminance* below a certain threshold. This way, only the bright parts of the image get a glow.
3. Create several copies of the thresholded texture and blur them with increasingly large Gaussian convolution kernels, e.g. 5x5, 11x11, 21x21 and 41x41, to smoothly expand the bright areas.
4. Add all the images together.

Exercise M6: Implement a *Bloom post-processing filter*.

Note 1: This algorithm can be heavily optimized. In fact, it is impossible to convolve with a 41x41 Gaussian convolution kernel and maintain an acceptable frame-rate. For real time implementations, an *approximation* is used [2].
Note 2: Some websites may imply that bloom can only be used when using high dynamic rendering (i.e. using a buffer that uses floating point numbers to represent color values, instead of an integer buffer that uses one byte per color and one for alpha), but you can definitely use the bloom effect when using a standard buffer (high-dynamic range is useful to trigger blooming for extremely bright pixels, far beyond “white”, as it would occur in realistic, high-end cameras and/or the human eye; what we do here with our 8-bit version is simulating a cheap cell-phone camera, so to speak…).

Reference image:

Bloom effect in short film Elephants Dream

References:
[1] http://en.wikipedia.org/wiki/Bloom_(shader_effect)
[2] http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/

[M7] God Rays (Volumetric lighting)

Category: *post-processing*

Volumetric lighting is a technique used in 3D computer graphics to add lighting effects to a rendered scene. It allows the viewer to see beams of light shining through the environment; seeing sunbeams streaming through an open window is an example of volumetric lighting, also known as *crepuscular rays* [1]. The term seems to have been introduced from cinematography and is now widely applied to 3D modeling and rendering especially in the field of 3D gaming.

In volumetric lighting, the light cone emitted by a light source is modeled as a transparent object and considered as a container of a "volume": as a result, light has the capability to give the effect of passing through an actual three dimensional medium (such as fog, dust, smoke, or steam) that is inside its volume, just like in the real world.

Details:

There are many different ways to implement the effect – from inverse shadow volumes to volume rendering. A nice post-processing method is discussed over here: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html

You can use this as basis for your implementation.

Exercise M7: Implement a *God Rays post-processing filter*.

Reference image:

References:

[1] http://en.wikipedia.org/wiki/Crepuscular_rays
[2] http://en.wikipedia.org/wiki/Volumetric_lighting
[3] http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=36
[3] http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html

HARD TASKS

[H1] HDR lighting and tone mapping

Category: *lighting / post-processing*

High Dynamic Range is a very simple technique that allows color channels to fall in a larger region than [0, 1] or [0, 255]. Only after rendering is completed, the high dynamic range colors are scaled down to the visible range using *tone mapping*.

Tone mapping is a technique used in image processing and computer graphics to map one set of colors to another in order to approximate the appearance of high dynamic range images in a medium that has a more limited dynamic range. Print-outs, CRT or LCD monitors, and projectors all have a limited dynamic range that is inadequate to reproduce the full range of light intensities present in natural scenes. Essentially, tone mapping addresses the problem of strong contrast reduction from the scene values (radiance) to the displayable range while preserving the image details and color appearance important to appreciate the original scene content.

Typically, the mapping is non-linear – it preserves enough range for dark colors and *gradually* limits the dynamic range for bright colors.

Note: Bloom and tone mapping are a great combination!

Reference image:

References:

[1] http://www.xnainfo.com/content.php?content=28
[2] http://expf.wordpress.com/tag/tone-mapping/page/2/

[H2] Deferred shading

Category: *lighting / performance*

In computer graphics *deferred shading* describes a three dimensional shading technique in which the result of a shading algorithm is calculated by dividing it into smaller parts that are written to intermediate buffer storage (called *G-buffer*) to be combined later, instead of immediately writing the shader result to the color frame-buffer. It is mostly used to do more efficient lighting calculations.

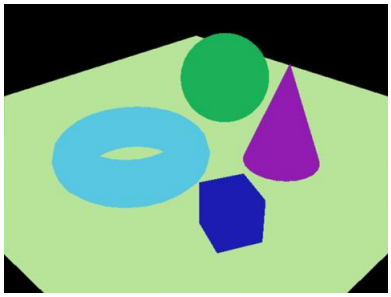
In your implementation so far, you render each object separately and apply lighting calculations to every pixel that gets produced, even if it is overwritten by another pixel later in the rendering process. This will take $O(M \cdot L)$ time, where *M* is the number rasterized fragments (visible or not) in the scene and *L* is the number of light sources.

In deferred shading, you put all the information you need for shading in a set of buffers, and you only perform the lighting calculations for the pixels that will actually be visible on the screen. This will take you $O(K \cdot L)$ time, where *K* is the number of pixels, plus some overhead of using separate buffers. It can easily be seen that this approach is more efficient for many light sources (for example 30 or more) and large overdraw. For a small amount of lights, the overhead of using the G-buffer usually cancels out the performance increase. In general, deferred rendering pays off for very expensive shading effects, and if objects are drawn in random order (front-to-back rendering would also avoid most of the overhead, as the pixel shader can be deactivated by the graphics hardware if occlusion is detected using an "early z-test").

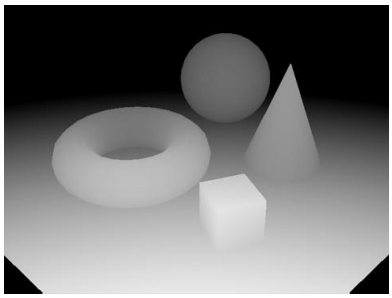
For complex shading, several buffers with different types of information are required. Implementations on modern hardware tend to use multiple render targets (MRT [3]) to avoid redundant vertex transformations. Usually once all the needed buffers are built they are then read (usually as input textures) into a shading algorithm (for example a lighting equation) and combined to produce the final result. In this way the computation and memory bandwidth required to shade a scene is reduced to those visible portions, thereby reducing the shaded depth complexity.

Note: to get full points for this exercise, an implementation with at least 20 lights should be shown.

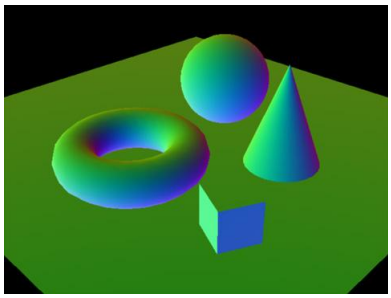
Example of the G-buffer:



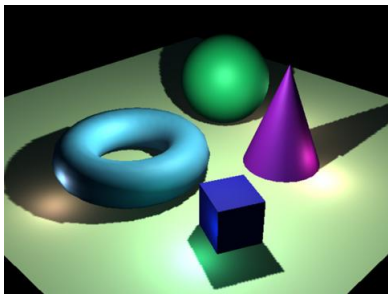
Diffuse color G-buffer



Z-buffer



Normals G-buffer



Final composition

References:

[1] http://en.wikipedia.org/wiki/Deferred_shading
[2] http://www.marries.nl/wp-content/uploads/2011/02/Comparison-of-multiple-rendering-techniques-by-Marries-van-de-Hoef-and-Bas-Zalmstra.pdf
[3] http://en.wikipedia.org/wiki/Multiple_Render_Targets

[H3] Parallax mapping

Category: *miscellaneous*

Parallax mapping (also called *offset mapping* or *virtual displacement mapping*) is an enhancement of the bump mapping or normal mapping techniques applied to textures in 3D rendering applications such as video games. To the end user, this means that textures such as stonewalls will have more apparent depth and thus greater realism with less of an influence on the performance of the simulation. Parallax mapping was introduced by Tomomichi Kaneko et al. in 2001.

Parallax mapping is implemented by displacing the texture coordinates at a point on the rendered polygon by a function of the view angle in tangent space (the angle relative to the surface normal) and the value of the height map at that point. At steeper view-angles, the texture coordinates are displaced more, giving the illusion of depth due to parallax effects as the view changes.

Parallax mapping described by Kaneko is a single step process that does not account for occlusion. Subsequent enhancements have been made to the algorithm incorporating iterative approaches to allow for occlusion and accurate silhouette rendering.

Note: Plenty of code samples can be found on the Internet.

Reference image:

References:

[1] http://en.wikipedia.org/wiki/Parallax_mapping
[2] http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/GDC06-Tatarchuk-Parallax_Occlusion_Mapping.pdf (2006)

[H4] Screen-space ambient occlusion

Category: *post-processing*

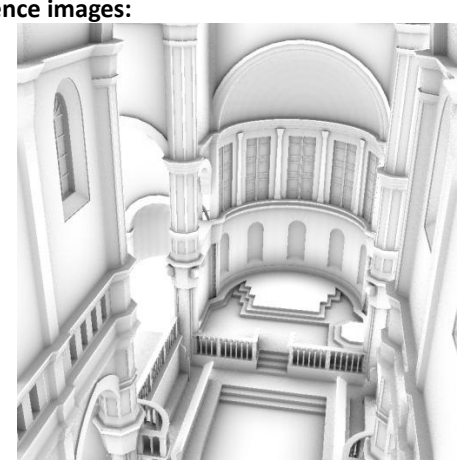
NB: this exercise is particularly ambitious.

Ambient occlusion is an approximate global illumination technique that illuminates a scene with a white spherical or hemispherical light source at infinite distance (such as an overcast sky; as an extension, colored environment-maps can be used as well, but white is most common). Unlike the local diffuse illumination, as discussed in M4/H5, ambient occlusion takes shadows into account. This can be implemented in a rather straightforward manner by using a large number of shadow maps to approximate directional light sources illuminating the scene from all over a spherical light source (btw: This is how the bunny logo was created, that is shown in the front-page of each lecture slide set!).

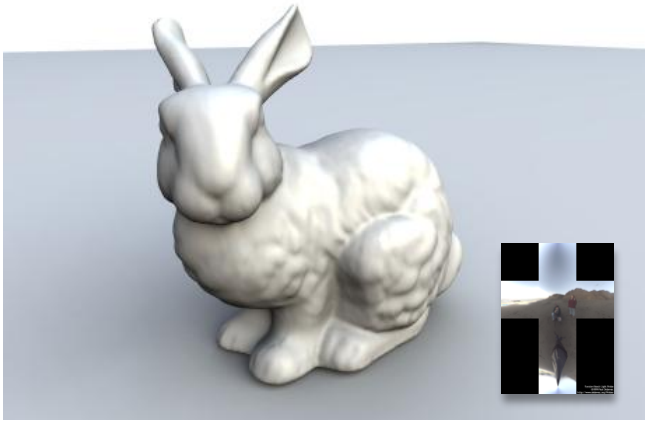
However, brute force, world-space ambient occlusion, as discussed above, is quite expensive. Rendering times are down to single digits FPS already for simple scenes, due to the many rendering passes with correspondingly high vertex and fragment costs. Screen space ambient occlusion is a neat hack to get a similar visual impression at much lower costs (and reduced quality, but the FPS are real-time friendly now, so this might be acceptable).

Screen Space Ambient Occlusion (*SSAO*) is a rendering technique for efficiently approximating the well-known computer graphics ambient occlusion effect in real time. The technique involves local sampling of the rendered scene to estimate if the pixel is shadowed from direct lighting by nearby geometry. It was developed by Vladimir Kajalin while working at Crytek and was used for the first time in a video game in the 2007 Windows game *Crysis* made by Crytek.

Reference images:



Only SSAO for shading (pure white light)



"real" ambient occlusion with an IBL light map – quite expensive though (just FYI; the assignment is on SSAO)
light map (c) Paul Debevec, from http://www.pauldebevec.com/Probes/

Normal shading *Normal shading + SSAO*

References:

[1] http://en.wikipedia.org/wiki/Ambient_occlusion
[2] http://en.wikipedia.org/wiki/Screen_Space_Ambient_Occlusion

[H6] Ray Tracing

Category: *core rendering*

Ray tracing is a rendering algorithm that uses ray optics to calculate light transport through each screen pixel. In practical asssignment 3, it replaces the rasterization rendering algorithm that you used for P1 and P2.

The brute-force ray tracing algorithm intersects each primary ray from the camera through a screen pixel with the primitives in the scene, evaluates a BRDF, optionally continues after hitting a mirror, or splits up in two rays after hitting a dielectric. Obviously, this is a slow process. This can be improved by using an acceleration structure.

For this assignment, you implement a Whitted-style ray tracer, with (at least) the following features:

- Models: your ray tracer must be able to render a static triangle mesh;
- Materials: at least diffuse, pure specular (mirrors), dielectrics (glass / water) with Snell and Fresnel;
- Lights: multiple point lights with distance attenuation and hard shadows;
- Lights: correct blending of the contribution of multiple light sources;
- Camera: your camera must meet the requirements described earlier in this document;
- Speed: an acceleration structure to speed up rendering.

Implementing texturing and texture filtering is not a requirement.

For the acceleration structure, you can either use a full-blown BVH, or a nested grid, see e.g.:

https://directtovideo.wordpress.com/category/realtime-rendering

How you visualize the output of your ray tracer is up to you. One option is to implement the ray tracer directly in a shader, which is then used to draw a screen-filling quad.

Grading:

Implementing the minimum requirements, and achieving at least 1 frame per second for a 10k polygon scene at 640x480 resolution results in an 8. For a 10, exceed the minimum requirements.

Reference images:

Screenshot from 5 faces by Fairlight & Cloudkicker

References:

[1] http://prof.johnpile.com/2013/11/05/realtime-ray-tracing-on-xbox360-via-hlsl-xna-c/
[2] http://dl.acm.org/citation.cfm?id=358882
[3] https://directtovideo.wordpress.com/category/realtime-rendering