COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

LIGHT TRANSPORT VISUALIZATION AND PERTURBATIONS

Baccalaureate Thesis

2014 Martin Pinter

COMENIUS UNIVERSITY IN BRATISLAVA

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

LIGHT TRANSPORT VISUALIZATION AND PERTURBATIONS

Baccalaureate Thesis

Study programme:	Informatics
Branch of study:	2508 Informatics
Educational facility:	Department of Informatics
Supervisor:	Prof. RNDr. Roman Ďurikovič, PhD.

Bratislava, 2014

Martin Pinter

87735085

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Martin Pinter
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov:	Light transport visualization and perturbations
Cieľ:	Prvý cieľ je naštudavať si metódy sledovania lúča a možnosti generovania svetelných ciest počas výpočtu transportu svetla. Druhý ciel je naimplementovať rozhranie kde si užívateľ zvolí začiatok lúčov a ich koniec a zobrazí všetky cesty v týchto oblastiach.

Vedúci:	prof. RNDr. Roman Ďurikovič, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	doc. PhDr. Ján Rybár, PhD.
Dátum zadania:	15.10.2013

Dátum schválenia: 21.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent vedúci práce

Acknowledgements

I would like to thank my advisor Prof. RNDr. Roman Ďurikovič, PhD.for his guidance and invaluable advice throughout my work on this thesis. I would also like to thank my colleagues from YACGS seminar for useful tips regarding the implementation. Finally, I want to thank my family and friends for their support and encouragement during my studies.

Abstrakt

Práca sa zaoberá algoritmami prenosu svetla a sledovania lúčov, pričom sa sústreďuje najmä na Metropolis light transport. Spomínané sú variácie tohto algoritmu, vychádzajúce z myšlienok pôvodného článku publikovaného v roku 1997. Predstavená je knižnica umožňujúca interaktívnu vizualizáciu algoritmov prenosu svetla v reálnom čase, ktorá tiež umožní používatelovi filtrovať jednotlivé lúče zvolením oblasti ktorou má lúč prechádzať. Taktiež umožní sledovať perturbácie zvoleného lúča podľa pravidiel MLT-algoritmu a jeho mutačných stratégií.

Kľúčové slová: Metropolis Light Transport, global illumination, vizualizácia, OpenGL

Abstract

Thesis revolves around methods of path-tracing and light-transport in computer graphics, with its main focus on Metropolis light transport. It lists through algorithms that spawned from the initial idea of MLT published in 1997, then presents a library for real-time interactive visualization of light-transport algorithms, that allows users to filter through light paths generated by the raytracer by selecting an area through witch the light path that he wishes to visualize must pass. The library also offers a visualization of perturbations according to the Metropolis framework and mutation strategy of the given algorithm.

Keywords: Metropolis Light Transport, global illumination, visualization, OpenGL

Foreword

While never being at the forefront of computer graphics research, the idea of Metropolis Light Transport has been kept alive since its first incarnation, with a decent number of works improving on the original notion. Lately, the interest seems to have resurged, with derivative works showing up on both SIGGRAPH 2012 and 2013. Yet, visualization tools are practically non-existent, not only for MLT based algorithms, but also for ray or path tracing as a whole (the best tool probably being a plugin for 3DS Max, designed primarily to allow light path manipulation for 3d artists). This work aims to round up the theory behind MLT, focusing on the original concept but also mentioning the various improvements, and to create a basic tool for visualization of light paths, and light path mutations. The core of this application will be relying on a tried and tested, widely used graphics libraries, namely Assimp and Mitsuba, with real time visualization in OpenGL. The reader should have at least a very basic background knowledge of methods in computer graphics.

Table of Contents

1 Previous works **2**

1.1 Metropolis Light Transport 2

1.2 Improvements over the original MLT 8

2 Lvis - light transport visualisation library **11**

2.1 External libraries and requirements 11

2.2 Compilation and usage 12

2.3 Implementation 12

3 Demo program **23**

3.1 Mitsuba data structure and path generation 23

3.2 Demonstration program implementation 24

3.3 Images and results 26

4 Conclusion and future work **28**

5 Manual - Lvis key bindings **29**

References **31**

List of Figures

2.1 Spherical coordinates, image courtesy of doc. RNDr. Miloš Božek, PhD. . . 17

3.1 A large mass of generated light paths. 26

3.2 Path selection. 26

3.3 Mutating a single path. 27

3.4 Virtual camera. 27

Introduction

The concept of simulating rays of light to create physically correct, believable and maybe even photorealistic renders of three dimensional scene is not a new one in the computer graphics community. If we take Arthur Appels work on ray casting as a basis, then the idea itself is almost half a century old. Over the years the concept developed and evolved into numerous algorithms and today most of them fall under an umbrella term of global illumination - a name indicating that these algorithms do not account only for light coming directly from a light source (such algorithms, including the original Appels raycasting, would be labeled under a direct illumination moniker), but also for light reflecting off of other surfaces in a scene, whether reflective or not. These include radiosity, photon mapping, ray tracing, path tracing, Metropolis light transport and other.

Today, global illumination algorithms are used in a variety of fields, from design and architecture to film industry. In this setting, every possible optimisation in terms of required time or processing power may have a serious impact on projects budget. And with these algorithms relying mostly on brute force, there are certainly places to improve. Since MLT may be one of the methods that pushes the boundaries again - and it has already successfully done so in the recent years - having tools to experiment with it should be beneficial.

Apart from providing a round up of theoretical knowledge in the field, this work is to provide a visualisation library for the MLT algorithm (and possibly for any other path tracing technique). The main challenge would be to visualise the data usually produced by an offline renderer in a real time interactive environment (OpenGL). The application also aims to be platform independent, using only widely supported, cross-platform libraries and methods.

Chapter 1

Previous works

In this chapter, theoretical knowledge of Metropolis Light Transport will be presented, independently from it's implementation used in the visualisation application. Variants and improvements are also discussed, presenting the path this method took after it's introduction in the computer graphics field.

1.1 Metropolis Light Transport

First proposed in 1997 by Eric Veach and Leonidas J. Guibas [VG97], MLT is a Monte Carlo method for solving the light transport problem, that seeks to improve the time complexity (which, in Monte Carlo evaluations of path-space light transport integrals, is generally closely related to the high number of samples that is required to avoid noise in the final render) by focusing the samples into regions of path space with high contributions to the final image. To do so, MLT calculates the amount of light (flux) flowing along the examined path, and uses this value in conjunction with Metropolis framework to determine whether it accepts or rejects the path. Acceptance probability is computed according to the Metropolis-Hastings algorithm and a new path is always created from the last accepted one using a set of predetermined mutation strategies. Therefore, light paths generated this way always form a Markov chain. In following sections, we'll attempt to explain the basic principles of this algorithm, as proposed in the original paper, hopefully in a more accessible way.

Metropolis Sampling

For every new path, that we get from a mutation run in the previous indentation of our sampling loop, we need to compute the acceptance probability. This is based on Image contribution function, which should, in theory, evaluate how visible is the effect of sampling and recording the information from a particular light path on our final render, so that we can accept the sample with probability proportional to its usefulness. Of course, this is something which is not easily estimated.

Fortunately, since we would use this function only to calculate our probability distribution in Metropolis-Hastings algorithm - which means we don't really need an absolute value of the function, just the distribution of it among the samples - we can use any function that is proportional to the aforementioned one. In our case, the Image contribution function will actually be the amount of light, or the luminous strength, flowing along current path.

The outcome of every mutation is based solely on the previous state, and therefore accepted states, which are in our case light paths, form a Markov chain - a random walk in state space, with predetermined probabilities for each transition. In return, we now know that the probability distribution, as the number of samples approach infinity, always converges to a single - stationary - distribution. The key is to generate and accept light paths as if they were sampled from such distribution.

To make our random walk behave like that, we will design the transition function in a way that the system satisfies the condition of *detailed balance*, or reversibility. That is, given the transition function $K(x|y)$, denoting the probability of transitioning into state x, when in state y, the condition $K(x|y) = K(y|x)$ must hold for every pair of states x and y. To achieve this balance for a particular pair of states (or samples), we first choose an arbitrary, *tentative transition function* T, for both $T(x|y)$ and $T(y|x)$. In MLT, T is given by a mutation strategy. To make the transition function reversible, we must also bear acceptance probability (which we'll define in next segment) in mind, along with the image contribution function f of the particular sample, that we need our probability distribution to be proportional to. To maintain equilibrium, it is sufficient that these densities are equal:

$$f(x)T(y|x)a(y|x) = f(y)T(x|y)a(x|y)$$

It can be proven that the reversibility attribute of a Markov Chain is equal to it producing samples according to stationary distribution. Furthermore, using this equation we can finally determine the acceptance probability. Since T is arbitrarily chosen and f can be calculated for each path, the only variable left in the equation is the ration $a(y|x)/a(x|y)$. In order to reach equilibrium as quickly as possible, we want to maximize both $a(x|y)$ and $a(y|x)$, therefore:

$$a(y|x) = min \left\{ \frac{f(y)T(x|y)}{f(x)T(y|x)}, 1 \right\}$$

The path integral formulation of light transport

The initial point of examination is the *light transport equation* between points x' and x'' given by

$$L(x' \to x'') = L_e(x' \to x'') + \int_M L(x' \to x'') f_s(x \to x' \to x'') G(x' \leftrightarrow x'') dA(x)$$

Here, M is the union of all surfaces, A is the area measure on M, $L_e(x' \to x'')$ is the emitted radiance leaving x' in the direction of x'',f_s is the bidirectional scattering distribution function (BSDF, this tells us how much of the light is reflected in a particular direction) and G represents the throughput of a differential beam between the point x and x'. We assume a geometric model of perfectly incoherent light, travelling in straight lines and being emitted, scattered and absorbed only at surfaces (although the formula was later modified in [PKK00] so that it accounts for participating media).

By introducing a *filter function* into the equation (expressing the amount of light that is received by a particular pixel - needless to say, this is zero for almost all light paths, expect for the ones which have points lying on lens edge) integrating over all points of the scene and then rewriting the formula into an integration over the path space, we get

$$m_j = \int_\Omega f_j(X) d\mu(X)$$

Here, m_j represents the amount of light received by pixel j , μ is the area measure and f_j the *measurement contribution function*, created by extracting the appropriate term (the one regarding the subpath we're currently examining) from the expansion of the integration over all points in scene (which we skipped for brevity, since paraphrasing of the original paper has already taken a large enough chunk of this work). X is a path from the path space Ω. Each integrand f_j can be written in a form

$$f_j(X) = w_j(X)f(X)$$

where $\mathrm{w_j}$ is again the filter function for pixel j and f represent other factors, same for each pixel on the image plane. Knowing this we can compute the estimated lightness of each pixel by sampling N samples according to some distribution p (using the acceptance probability from the previous section) and using the identity

$$m_j = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{w_j(X_i)f(X_i)}{p(X_i)}\right]$$

The only missing part we are now left with is the probability distribution $p(X_i)$ or more precisely, since

$$p(X_i) = \frac{f(X_i)}{\int_\Omega f(X_i)d\mu(X_i)}$$

we are actually looking for a total amount of radiant power received by the image plane.

While our transition function maintains detailed balance, the samples themselves will have desired distribution only as $i \to \infty$. This problem, also known as *startup bias* in various kinds of importance sampling methods, is most easily avoided by starting from an approximate equilibrium distribution computed using another sampling algorithm (like bidirectional path tracing). The initial path in the Markov chain is then chosen from the paths generated this way (which are called the seed), with probability proportional to its contribution function - thus, even the first state is, statistically speaking, sampled from the stationary distribution. In [VG97], it is suggested to restart the process multiple times with different seeds (each time contributing to the same image).

Putting it back together

To sum up all the mathematics - we fire up the algorithm by collecting a number of samples using a different method, like bidirectional path tracing, to get approximate sample distribution. We then choose an initial state and continue with our Markov chain random walk until we're happy with the result. Path mutations are used to advance to the next state and in each step we have all the parameters we need to calculate both acceptance probability and expected luminance of a pixel (1.1).

It should be noted that so far the equations presented account for monochrome images only. Yet, expanding into colour is straightforward - we simply sample for each

Source 1.1 Pseudocode of the MLT agorithm, as specified in [VG97]

```
x = GenerateInitialPath()
image = EmptyImage()
for i=1 to N
        y = Mutate(x)
        a = AcceptationProbability(y|x)
        if RandomValue() < a
                x = y
        RecordSample()
return image
```

part of the desired colour spectrum (like RGB) separately, with image contribution function being the luminance of the sampled spectrum.

Path Mutations

The main disadvantage of MLT lies in the strong correlation of consecutive samples. While this could be beneficial when exploring areas of path space that contribute largely to the final render, it can also mean that the algorithm can get stuck on an unimportant path, having trouble to move on from it. While this is already less likely than the occurrence of the former phenomenon (since we're sampling proportionally to paths importance), we can minimize the chance of it by choosing a wide-enough array of path mutations. And while the vicinity of the less luminous paths may not need as many samples as the one of lighter paths, we still need to sample from all paths that contribute to the final image, otherwise the render will be noisy. In this section we present the mutation strategies used in [VG97].

Bidirectional mutations

Perhaps the most straightforward of all, bidirectional mutations allow us to make both small changes, or to throw away the whole path and start with a completely unrelated one. Though the chance of the later is slim in comparison to other possible mutations, it is an important tool in ensuring of the *ergodicity* of a random walk - to satisfy it, it is sufficient when for every state there is a non-zero possibility of transitioning to any other state. That ensures that regardless of the initial sample, the random walk converges to the same ergodic state - the stationary distribution.

When the algorithm chooses to perform a bidirectional mutation, it first chooses a length of the subpath that would be deleted (in terms of number of vertices). Shorter subpaths are preferred - this ensures both high acceptance probability and low cost of a mutation, both of which are requirements for a good mutation strategy. It then chooses two vertices on the path, with distance (naturally, again in terms of vertices) equal to the one previously calculated. Subpath between these two is then removed and replaced by a new one (preferably, but not necessarily, with the same length as the old one). This is done via bidirectional path tracing - one branch starting in first, the other in second vertex, both reaching desired length and then finally connecting. If the path is obstructed, i.e. the final interconnection passes through a wall, the mutation is immediately rejected and a new one is generated.

Perturbations

Perturbations are MLTs main tool for exploring small areas of high importance. The idea is to take some of the vertices of a light path and move them slightly, thus making a less drastic change to the path then by most of the bidirectional mutations. Each vertex is moved by a random distance R in a random direction ϕ. The angle ϕ is chosen uniformly, while R is exponentially distributed between two arbitrary values $r1 < r2$.

If the perturbation causes any of the vertices to change the kind of surface they are on, or the light path becomes obstructed, the mutation is automatically discarded. Three kinds of perturbations are proposed in the original paper. We'll use Heckbert's regular expression notation, for denoting surface properties - S, D, E and L stand for specular surface, non-specular (diffuse) surface, lens edge and light source respectively, + and * signs for "at least one" and "zero or more".

Lens perturbations

We delete subpath of the form (L|D)DS*E, then reconstruct it starting from lens edge, towards the rest of the light path (whether it continues with a non-specular surface or consists just of the final vertex on a light source). Notice we need the last vertex that is to be modified to lie on a non-specular surface - this allows us to connect the end of the newly generated path to the previous one, since we have more freedom with choosing the direction of a ray that is scattered on non-specular surface, in comparison to specular surfaces where the BSDF is more strict.

Caustic perturbations

Because of the subpath form that is imposed on us when doing Lens perturbation, we now need a different technique for exploring neighbourhood of light paths causing caustics, since these are of the form LS+DE. Caustic perturbation starts off by deleting the subpath of the form (D|L)S*DE and then reconstructs from vertex closest to the light source. Thus, the last vertex of a new subpath lies again on a non-specular surface, which allows for easy connection with lens edge.

Multi-chain perturbations

Neither of the aforementioned perturbations work on paths with suffix of the form (D|L)DS*DS*DE. This can be handled by chaining perturbations, starting and ending on non-specular surface (or lens edge or light source).

Lens subpath mutations

Lens subpath mutations help with stratification of the samples over the final render and also with lowering of the cost of the whole algorithm. The idea is to delete the lens subpath - (D|L)S*E - and then, with each new mutation, replace it with a subpath aimed at a different pixel on the image plane. This way, the rest of the path is used to it's fullest and not thrown away at the time it could've still been useful.

1.2 Improvements over the original MLT

The interest in Metropolis Light Transport seems to have resurged in recent years, with new theses expanding on the idea of statistically aided Monte Carlo light transport solver. In this section, we will briefly introduce the more prominent ones, ordered by the date they were published, starting from the earliest.

Metropolis Light Transport for participating media

In 2000, MLT was extended to account for participating media. This was done by expanding the path space to include media interactions and then alternating between scattering and propagation events. Scattering event, just like in [VG97], chooses a new direction at a point of interaction, while propagation event calculates the distance travelled along a path segment passing through a participating media, before it is scattered. This is done according to transmittance of the medium. If the segment doesn't interact with participating media, the propagation step always chooses the first surface point intersecting with the ray, thus behaving like regular MLT.

The paper also introduced new kind of participating media related mutations - the propagation perturbations - which displace the point of interaction along a certain ray segment.

Improved and unified mutation strategy

The paper starts up by showing, how can the correlation of Markov chain path tracing algorithms lead to increase in integration error, also showing that it's impossible to use a single mutation strategy in path space, without increasing the correlation. It continues to solve this problem by performing mutations in a different, *primary sample space*, which is essentially an infinite-dimensional cube, from which a point in space in sampled, and values gathered from the vector defining this point are then used as inputs for desired path length and BSDF functions at path vertices. Note that even when the vector is infinite in theory, we only need a first couple of values, which are the only ones that will be generated.

The vector is then mutated, as if we were changing the position of a point in the sample space, and every time we need to look at an additional (not yet generated) dimension, this one is mutated the same amount of times as the rest of the vector was. Since this could lead to massive slowdowns in later stages of the path generation, *large steps* are introduced, which are in a way equivalent to bidirectional mutations that completely replaced the path. These sample a new point in the n-dimensional cube and reset the mutations - so that when a new dimension is needed, it only needs to "fall back" towards the last large step. Large steps also help in ensuring of the ergodicity of algorithm. The method therefore uses only two kinds of mutations, instead of the usual array of four or five, while generally improving the efficiency.

Gradient-Domain MLT

Another global illumination algorithm, partly based on Metropolis inspired weighted sampling was published in 2013. Contrary to Veaches implementation, samples are not concentrated around the most luminous paths in the path space, but around areas of largest gradient in image space. This way, the algorithm first generates a rough, noisy image, which provides it with enough data (in from of a 2D image) to use the Poisson solver to fill in the gaps. Furthermore, even if this technique may appear as a too big of an approximation, it was proven that the algorithm is actually unbiased.

Chapter 2

Lvis - light transport visualisation library

We now present a C++ library that allows for selective visualisation of light paths generated by Metropolis Light Transport, or any other path or ray tracing algorithm. It allows the end user to control the volume of visualized data, provides control over the real-time scene visualisation in terms of camera positioning and lightning, supports interaction with mouse and keyboard, provides tools for path vertex position based filtering, mouse selection and visualisation-to-pathtracer feedback.

Since it was develop with MLT in mind, it also supports a set of features closely related to this method, namely the capacity to visualise a sequence of paths while highlighting mutated segments, and to provide the path tracer with a feedback in form of a user selected light path, which can than be used as a new basis for the next mutation. The aim of this library is to provide a tool for better understanding of existing mutation strategies and possibly also for development and debugging of a new ones.

2.1 External libraries and requirements

The library is written in C++, with the visualisation being handled by OpenGL, using the 3.0 standard - this being the highest OpenGL standard currently available to many Linux distributions and graphics cards (graphics card drivers in particular are the main issue here), or in other words, the highest version supported by the machine this library was developed on. As with most OpenGL applications, we rely on a set of additional libraries to provide us with the more standard functionality - GLEW to allow for hassle free use of the modern parts of OpenGL, GLM (OpenGL Mathematics) to provide us with the usual vector and matrix operations, SFML (Simple and Fast Multimedia Library) for context creation and input handling, DevIL (Developers Image Library) for texture loading and finally Assimp (Open Asset Import Library) for model loading. All of these are freeware, open source and cross-platform.

2.2 Compilation and usage

If you wish to compile and run a program using the Lvis library on your local workstation, you'll need all of the libraries mentioned in the previous section, along with their respective dependencies, installed. OpenGL 3.0 along with GLSL 1.3 support is also required. The enclosed Cmake file was tested on Ubuntu 14.04, but should work on any Linux distribution, and may require some minimal modifications for Windows. Your compiler must also support the C++11 standard.

In the path tracer (or other program you wish to use with Lvis), you first need to include `lvis/include/lvis.hpp` and create a new instance of Lvis object (with arguments to the constructor being path to scene file, and path to scene file textures). Then, whenever you wish to start the visualisation, you just need to call the run() method on your Lvis object. This creates the OpenGL rendering context in a new window, starts the visualisation and allows you to safely access all of the other public methods provided by Lvis. These will be explained in the subsequent sections (for correct usage, it is recommended to read at least the sections on threading and the section on light paths), with the example program provided in Chapter 3.

2.3 Implementation

Class Lvis

The main class provides all the interfacing, between both the Lvis library and the path tracer, and user input and all other library classes. It holds all of the data structures (with pointers to the instantiations of other classes), and also contains methods for selection and path filtering. In this subsection we provide a categorised rundown of its functionality.

Initialization

While the constructor code only assigns paths to scene files to their respective variables, the initialization of OpenGL and member data occurs when `Lvis.run` is called. Context creation is handled by SFML, which has a rather streamlined approach towards it - the downside is having little control over some of the more advanced OpenGL settings, which we don't really need anyway. All of the shader files are loaded, compiled and linked at this point, and Assimp is used to parse every 3D model needed for the visualisation (more on this later). If there's an error at any stage of the initialization, program prints a warning and exits.

Threading, flags and public methods

We take advantage of the C++11 standard and implement threading using the `std::thread`, `std::mutex` and `std::condition_variable`. Multithreading is used so that the user can freely navigate and interact with the scene even while new light paths are generated. Data modifiable by both threads (expect for some of the flags, which we'll cover soon) are protected by a `std::unique_lock` on a single mutex object, and condition variable is used to signal the path tracer thread after each draw cycle. The rest of the cycle being again protected by the mutex - since each of the functions may require an access to shared memory, and we're not too concerned about the time efficiency (and in fact, multiple lockings and unlockings may hinder it even further), it's much clearer to simply lock the whole block and allow path insertions only in-between renderings.

Communication between the two threads is provided by a set of public methods and also via *flags* (boolean variables). The only threading-related method not associated with these flags at all is `Lvis.join`, whose use in the context of threads is fairly self-explanatory. The same can be said about the `flagRunning` and its related methods, while `flagSelected` and `flagSingle` simply signalize to the second thread that a new path was selected and should replace the current active one, or that we want to be pushing and visualising paths one after another, using the method `pushSinglePath` instead of the standard `pushPath`, respectively. The `flagPause` is a bit more interesting, and it's usage is left entirely on the end user. The idea, and proposed design, behind it is for it to be used in conjunction with `Lvis.wait` - which pauses the execution of a thread it was called from until signaled by `flagWaiting`. Since `flagWaiting` is private to prevent race conditions, we provide the user with `flagPause` to signal the path tracer thread whether it should call for `Lvis.wait`. An example usage (but not the *only* right way to do so) of this flag is show in the demonstration program (3.2), where it also provides different behaviour in conjunction with flagSingle.

Apart from `Lvis.clearX` methods, which are again self-explanatory, we've yet to mention a couple of functions for further control over the visualisation and a set of methods limiting the data throughput. `Lvis.setupLight` allows us to position a single point light source, to allow for a closer match between lightning in visualisation and in the offline rendering. `Lvis.setupVirtualCamera` positions a camera model, facing a certain direction with the given up-vector, to match the camera used in the rendering.

The last three of the public methods are `Lvis.setLimit`, `Lvis.setDrawLimit` and `Lvis.setThreshold`. As their name imply, each of these modifies a certain parameter - or a certain limit - of Lvis class. `limit` is the maximum number of paths that can be stored inside `std::vector inputPaths` (those are the paths waiting to be filtered and drawn), `drawLimit` is the maximum number of paths drawn in the scene at each point in time and finally, `threshold` is the minimal size of `inpuPaths` before it can be filtered and subsequently drawn.

Input, selection & colour picking

All of the keyboard and mouse inputs are handled by SFML. This is done in two ways,the first one is through the `sf::Event` class, which polls the window class for all mouse, keyboard and window events (including window resizing or closing). The advantage of this approach is that only inputs that happen inside the window (or while the window is focused) are registered, but on the other hand, the keyboard events are triggered only when the key is pressed or released. To implement continuous camera movement (via keyboard), we use the `sf::Keyboard` class, which allows for real-time feedback, but we need to check for window focus on our own.

Left mouse button clicks triggers the selection mechanics - we need to differentiate between clicking one of the axis of the active sphere selector, other selector, light path and any other place on the scene. This was implemented through the use of off-screen rendering and colour picking. For axes it's pretty straightforward, since we're reading `RGB` values and each axis occupies only a single channel. Each of the spheres and paths have a unique id, which is assigned using the `counter` variable and `Lvis.counterUp` method[1]. This id is then "hashed" into an `RGB` colour (this idea was taken from [Bub]), which is then used to draw the object during the picking stage. Spheres are, of course, drawn as solid objects instead of wireframes during this step. The picked colour is then used as a key value for the relevant map structure.

Source 2.1 Encoding index into RGB color for picking.

```
r = index&0xFF;
g = (index>>8)&0xFF;
b = (index>>16)&0xFF;
```

Selector positioning and movement

Every new sphere selector is created at the position of virtual camera (defaults to scene origin) - this seems like a good place to start since it's also the place where all of the paths generated by bidirectional tracers will end. While the actual translation, in terms of modifying its Model matrix, is done inside the `SphereSelect` class, the values passed to it are determined in the `Lvis` class, therefore we now present their calculation to you.

When the selector is active (i.e. it has been clicked on), it displays coloured arrows along each of the three axes of the object coordinates (we've actually nicknamed these arrows as "axes" in the source code and already referenced them like that in this paper, so might as well carry on with it). The green one is always oriented towards the positive y-axis, while the other two switch from positive to negative depending on the camera position (to allow for easier manipulation).

When dragging the sphere along one of the axes, the direction is decided this way - first, we use the same Model - View - Projection (MVP) matrix that we've used to

[1]Note that this means there is no overlap in the ids of spheres and paths, and since we're clearing the colour buffer each time we're selecting a new object, we could have used different counters for different shapes, allowing us to draw slightly more paths at once. But since the number of used selectors will always be much smaller than the number of drawn light paths (unless we choose to draw a small number of paths, in which case this is of no concern at all), we use a single counter for convenience.

display the sphere to get the window coordinates of our vector. This transformation is applied to a vector of homogeneous coordinates, where the w (the fourth value) is set to *zero*. This way, all of the translations contained in the MVP matrix are omitted, and the vector we get from this transformation will have it's origin in the center of the screen, pointing towards one of the four quadrants. Based on this information, we either translate along the arrow, when the mouse cursor is moving towards that quadrant, or in the opposite direction if it's moving away from it. This is done separately for the x and y directions, so for example, if the transformed vector lies in the first quadrant, dragging it both upwards and to the right side moves the sphere along the arrow. The same applies for scaling.

Drawing

All of the actual drawing (i.e. the binding of VBOs and calling `glDrawElements` or `glDrawArrays`) is performed in other classes, `Lvis.draw` simply iterates through all of their instances and calls the appropriate methods. It is also responsible for setting the uniforms for camera position (that is, the position of actual OpenGL camera, for use in the Phong shader) and light position, and choosing the appropriate shader for drawing the scene.

Camera

In modern OpenGL, we're no longer forced to use the built in ModelView and Projection matrices. Instead, developers are encouraged to implement these matrices by themselves, with GLM becoming the standard library for creation and manipulation of both the matrices and the vectors they are transforming. The primary function of `Camera` class is to hold the View and Projection matrices, and provide functions for their manipulation. Because of our grouping of these matrices in a single class, we commonly retrieve a single *ViewProjection* matrix (using the `Camera.getVP` method), and then use the positional attributes of the objects in scene to construct a model matrix (if needed).

Camera can operate in two different modes, the default one is always facing the center of the scene, while the *free mode*, as the name suggests, allows for free movement around the scene and rotation around the camera origin. Rotation is represented in spherical coordinates (fig. 2.1, the difference being that we use `xi` and `fi` in place of θ and φ,

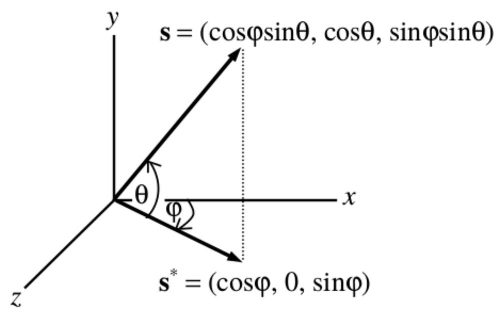

Fig. 2.1: Spherical coordinates, image courtesy of doc. RNDr. Miloš Božek, PhD.

respectively). Since in `Lvis` class, this rotation is performed by dragging the mouse, and only the most recent position values are sent, we need to remember the angles set before the rotation began, which is done through the use of `Camera.memorizeSpherical`.

In the default mode, the unit vector calculated from these angles is multiplied by the desired distance from the center and then used to position the camera, while in the free mode, the position is controlled via the keyboard input, and the unit vectors are used to set the viewing direction. We use `glm::lookAt` method to compute the View matrix in both approaches, along with `glm::perspective` to construct the desired Projection matrix.

Shaders

Before introducing the drawing procedures, let us present a quick rundown of the shader programs used by the library. `lineProgram` is the simplest one, only performing the standard Model-View-Projection transformation in vertex shader, and colouring each fragment with the colour passed as a uniform (same for each fragment in the whole mesh). As the name indicates, it's used primarily for drawing polylines (light paths), but also for drawing sphere selectors and axes, since no further functionality is required there. For scene (and virtual camera) drawing, one of the three shader programs is used - `basicProgram`, `noTextProgram` or `textOnlyProgram`. They all share common vertex shader and data structure, therefore differentiating only in fragment shader code. The basic shader combines texture sampling with Phong model ambient,diffuse and specular contributions, `noTextProgram` omits the texture and similarly, `textOnlyProgram` forgoes the lightning calculations. The user can switch between these three at runtime.

Phong illumination model

Implementation of the fragment shader was taken from an online tutorial [OGL] - since the computation is given by a single, standardized formula, no changes were needed. This formula (in our implementation, which uses only a single light source, and we also omits most of the material constants, using the same formula for all objects and materials) goes as follows

$$I_p = i_a + (L.N)i_d + (R.C)^\alpha i_s$$

where i_a, i_d and i_s are vectors containing the ambient, diffuse and specular components of lightning (in our case, they are set to `(0.05,0.05,0.05)`, `(0.7,0.7,0.7)` and `(1.0,1.0,1.0)` respectively), α is the shininess constant (again, set tu `4.0` for all the materials) and I_p is in our case the resulting brightness of a fragment (since our i_a, i_d and i_s are all represented as intensities of white light). Rest of the equation consists of normalized vectors and their dot products, where N is the surface normal, L is the direction towards the light source, R is the direction that a perfectly reflected ray of light would take and C is the direction towards the camera, all of these pointing from the currently processed surface point. In the `basicProgram`, this value is then multiplied by a value sampled from the appropriate texture at given UV coordinates. In `noTextProgram`, the whole calculation is omitted.

Assimp, sceneLoader and meshes

Let us first provide you with a brief note on Assimps internal data format. After loading the model file with desired flags - in our case, we're flagging to triangulate (since our OpenGL standard supports only triangulated meshes), calculate tangent space and calculate normals (both for the use in Phong shading model) - we are left with a `aiScene` object. This object contains a pointer to `aiNode` tree structure, with each node containing a certain set of data, out of which we're only concerned for an array of meshes. Therefore, we traverse the tree structure and parse every mesh into our mesh class format, processing all the vertices, indices, textures with its coordinates, normals and tangents (in `sceneLoader`), and afterwards creating VBOs (for vertices and indices) and binding the data to them (in mesh class). Both of these classes were again taken from an online tutorial [OGL], and only slightly modified to our needs - i.e. the original file used SDL for texture loading - instead, we opted to use DevIL as a more lightweight alternative - and we also added an option in the `sceneLoader.draw` and `mesh.draw` functions to draw wireframe model instead of a solid one (for use in sphere selectors).

When drawing the scene, first the shader program is chosen and uniforms containing light and camera positions are set. All of this happens in the `Lvis` class, which then calls the draw function of `sceneLoader`, along with the shader program id and ProjectionView matrix gathered from the `camera` class. The `sceneLoader` class simply calls the `sceneLoader.draw` function of all of the meshes associated with it, delegating the data passed to them from the previous class. And finally, actual binding of vertex data and drawing (call to either `glDrawElements` or `glDrawArrays`) happens in the `mesh` class. We are using `drawArrays` insted of `glDrawElements` when drawing wireframe to conserve a bit of data transfer between CPU and graphics card - since we're already doing a lot more of it by replacing a single call by one for every triangle. The drawing of virtual camera, spheres and their axes looks almost exactly the same, except that we add another step inside these classes to allow for further modifications to MVP matrix and shader uniforms.

Sphere selectors

Already mentioned a couple of times in this paper, `sphereSelector` class allows us to filter through the drawn, and about-to-be-drawn light paths. The filtering is done based on the distance from the center of the selector, therefore they are visually represented by a sphere (iconosphere wireframe in particular, since again, we're only allowed for triangulated surfaces in the newer OpenGL, and we want the model to be wireframe for practical purposes). The spheres are all created with the radius of `1.0`, which is then (together with the actual area of effect) controlled by the variable `scale`.

At this point, we should explain how the scaling and translation works from inside of the `SphereSelect` class (the way that the values for these operations are passed was already discussed in the `Lvis` class subsection). We use a slightly different system than in `Camera` class, but in the end, both of them are practicaly the same thing, just taken from a different angle. While with the `Camera` we saved the `oldXi` and `oldFi` at the start of the operation, here we have `glm::vec3 tempTrans` for translation and `tempScale` for scaling, which are rewritten by every call of `SphereSelect.translate` and `SphereSelect.scaling` respectively, and are only written into their "permanent" counterparts when `SphereSelect.finishTranslate` and `SphereSelect.finishScaling` are called. The downside of this being that we need to use both the temporal and the permanent variable to the transformation matrices, to provide the user with feedback.

All of the actual filtering is done via the function `pointInside`, which calculates the distance between the center of the selector and the point passed as a parameter (usually a position of a `PathVertex`), and returns a boolean value based on the `scale` and the result of this calculation. The library provides four basic kinds of selectors, three of them - the green (start), red (end) and yellow (any) selector work as if all of the selectors of the same kind made a logical statement containing only binary `OR` directives - therefore, if a single path vertex lies in any one of the selectors (still talking about one kind only), the whole statement is resolved as `TRUE` and the path is drawn. Similarly, we have one more kind of selector, the white one, which works as if all of them were in a logical statement containing only binary `AND` - so that path is only drawn if at least one vertex lies in every single white selector in the scene. The implementation of this algorithm can be found in the `Lvis` class (as a part of the `Lvis.filterPath` method).

Since spheres are to be selectable by a mouse click, this class also contains methods for transforming index number into `RGB` colour and vice-versa, for the use in picking algorithms, again located in the `Lvis` class.

Light paths

Each light path is represented by a single instance of its class, which in turn consists primarily of the vector of pointers to `pathVertex` instances, which store a single `glm::vec3` object with vertex position. There are two reasons for having vertices as separate classes - the primary one during the development was the ability to swiftly create a randomized light path for debugging purposes, and the reason they've stayed in the later stages of the development cycle being the delegation of `PathVertex.getInfo` and the overall tidiness of the code.

The light path is set up by appending new vertices from the start towards the end (note that sphere selector differentiates between the two boundaries). When we are ready to visualize the path, we must first call the `LightPath.finalize` method, to generate and bind a new VBO for the path. If we're doing this repeatedly, the previous VBO buffers are correctly deleted. If we try to add a new vertex after finalizing the path, it will be again marked as incomplete and won't allow us to visualize it before calling `LightPath.finalize` on it again. This method also creates an info log about the path, that we can then retrieve as a string.

Since we're adhering to the OpenGL 3.0 standards, where geometry shaders were still not the part of the core specification, and `glLineWidth` was not yet considered deprecated, we're being lazy and using this function to give our paths desired width, which can be set by the end user. Just like `SphereSelect`, `LightPath` contains methods for determining pick colour.

Virtual Camera

The final touch to the visualisation is in adding the model of a camera to a place where the camera of the offline renderer is positioned in the scene. This is set via the method `VirtualCamera.lookAt`, which takes the same parameters as Mitsubas, and probably also other renderers, lookat xml directive - therefore, if we're using our library in conjunction with such library, it is straightforward to correctly setup our virtual camera model.

`VirtualCamera.lookAt` uses another method to help with the correct positioning - `VirtualCamera.faceVector` - which takes two vectors as an argument, and returns the rotation matrix, that transforms the orientation of the first one to be the same as that of the second one. This is done by normalizing the vectors, after which their dot product will be equal to the cosine value of the angle between them, and then computing the cross product to get the axis perpendicular to both of them, around which we, finally, perform the rotation.

To get the model, which was originally facing towards the positive infinity on the z-axis, with up vector lying on the positive part of the y-axis, to look at the desired position, we first need to obtain the direction we want it to be facing - by subtracting the `from` position vector from the `to` vector. We can ten perform the rotation (provided by the `faceVector` method) from the default facing to the new one. We also perform the same rotation on the default up vector - to find out what will the up vector look like after this transformation. From that we get the new `tmpUp` vector, and we need to again find a rotation matrix that transforms it to our desired up vector - which we'll do again by the same method as before. As a last step, we translate the camera model to the desired position, and apply all of these operations to the ViewProjection matrix of the OpenGL camera. Also, before any of these transformations , the whole model is scaled down and translated so that it's lens are at the initial model coordinates origin - this is the place were all of the light paths will end.

Chapter 3

Demo program

We present a simple test program to demonstrate the functions and capabilities of Lvis library. To do so, we've used Mitsuba, a physically based renderer developed by Wenzel Jakob. Apart from LuxRenderer, it's the only one of the larger open source renderers which supports Metropolis Light Transport. The downside is that Mitsubas plugin oriented program structure (described below) makes the usage of our library (which requires access to light paths and their vertex data) rather complicated and impractical. Instead, we opted to use it separately, utilizing it's path space MLT algorithm together with its built-in path print function to generate a mass of light paths, which we then stored in a text file and later parsed into our demonstration program. This, of course, prevents us from showing the feature of selected path feedback, but with the rest of the visualisation qualities unaffected - and probably even a minor speed up in gathering of each new light path - that's just a minor setback.

3.1 Mitsuba data structure and path generation

For the purposes of our demonstration, the only segments of code from the Mitsuba sources we were interested in were the ones regarding the light path structure. With the core functionality split between four libraries - one of which (`libbidir`) was containing the support framework for all bidirectional path tracing algorithms - it was straightforward to locate the data structures of Mitsubas internal path format. Additionally, this class already had a method of printing it's data into a single `std::string`.

Next, we needed to find the MLT mutation and rendering cycle, from which these paths are instantiated and mutated. All rendering algorithms in Mitsuba are handled as integrator plugins - the plugin part meaning that they are loaded and compiled on demand at runtime, and the integrator moniker implying that at heart, these algorithms perform integration over a high-dimensional space. Each of these integrators contain a *work unit*, which performs all of the heavy lifting and also allows for parallelization, with multiple work units each solving a different subset of the integral. Therefore, the only file we've needed to modify was `mlt_proc.cpp`, containing the worker unit class for the MLT algorithm.

Our changes were fairly minimal, only adding an output `filestream`, which we then supplied with the output of `print` function of the `current` path pointer. Our program is able to correctly read and parse any textfile consisting of data aggregated using this function. Note that we're visualizing only the *accepted* light paths, and rejection of the proposed mutation means that subsequent light paths will be identical. This is not viewed as a problem, since it keeps a record of rejections in our visualisation (without the need to parse all of the proposed paths). Since we've only used core c++ functionality in our modifications, there was no need to tamper with the `.cmake` configuration. Apart from the light path data, Mitsuba also provides us with a nice set of statistics from the rendering(3.1 - we have omitted the information on texturing since it's not relevant to this work).

3.2 Demonstration program implementation

Apart from the main loop (3.2), this quick demonstration contains only functions for parsing the Mitsuba generated paths file. `parseNext` appends the vertices of the next path in the file to the `current` light path, and uses `parseVertex` which is self explanatory. We ignore all of the data except for the path vertex coordinate. Because of this, we've managed to use a slight modification of this program to generate files where all the unnecessary data was omitted, and only vertices and new path markers are left. These files are still compatible with all of the previous parsing methods. Before starting the visualisation, we parse and throw away a random number of paths, so that we get different results when running the program multiple times.

In the main loop, after successfully parsing and finalizing a new path, the program decides what to do next based upon current flags set by the Lvis library. `flagSingle` signali-

Source 3.1 Statistics from the scene rendering, using the MLT algorithm

```
* Bidirectional mutation :
  -  Acceptance rate : 10.33 % (58.69 K of 567.95 K)
  -  Successful generation rate : 21.22 % (120.54 K of 567.95 K)

* Caustic perturbation :
  -  Acceptance rate : 73.91 % (435.24 K of 588.88 K)
  -  Successful generation rate : 83.50 % (491.72 K of 588.88 K)

* General :
  -  Normal rays traced : 14.987 M
  -  Shadow rays traced : 4.808 M

* Lens perturbation :
  -  Acceptance rate : 73.76 % (434.52 K of 589.09 K)
  -  Successful generation rate : 84.31 % (496.66 K of 589.09 K)

* Path Space MLT :
  -  Accepted mutations : 52.56 % (928.45 K of 1.77 M)
```

Source 3.2 Main loop of the demonstration program

```
current=new LightPath();
parseNext();
...
//handle end of input file...
...
current->finalize();
if (elvis->flagSingle) {
        elvis->pushSinglePath(current);
        if (elvis->flagPause) elvis->wait();
        continue;
}
if (elvis->flagPause) {
        elvis->flagPause=false;
        elvis->wait();
}
elvis->pushPath(current);
```

zes whether we should send the generated path through the method `Lvis.pushSinglePath` or the regular `Lvis.pushPath`. In our setup, we use the `flagPause` in conjunction with `flagSingle` - if the latter is true, `flagPause` signalizes the main thread should wait every time it pushes the new path. If we're using `pushPath` instead (i.e. `flagSingle` is false), `flagPause` indicates that the main thread should wait, even if the input buffer is not yet full (and the `drawnLimit` has not yet been reached). The loop continues either until reaching the end of file, or until the visualisation is ended by the user.

3.3 Images and results

We present sample images, captured while using the demonstration program.

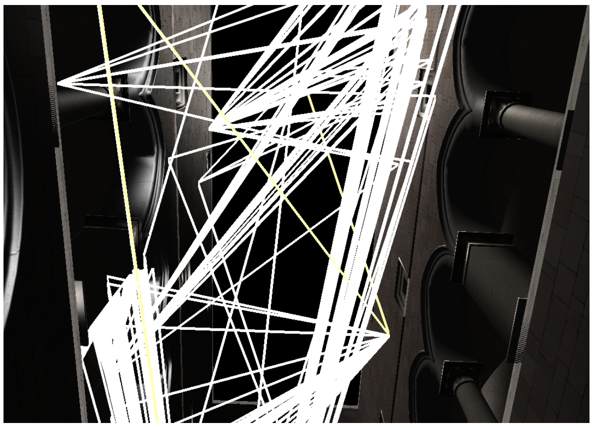
Fig. 3.1: A large mass of generated light paths.

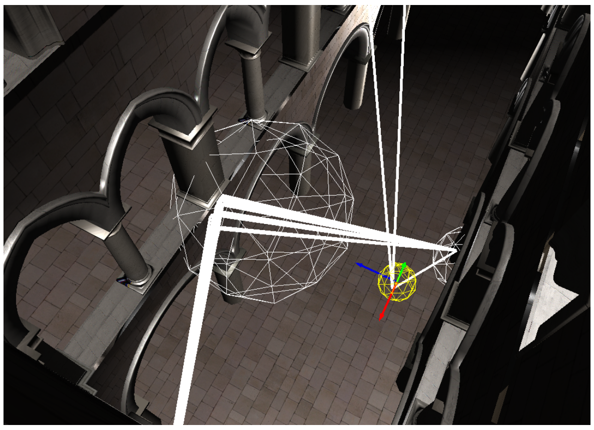
Fig. 3.2: Path selection.

The program functions as expected. Yet, if we were to pinpoint it's biggest weakness, it would be in filtering of `inputPaths` when using selectors. Since we have to check every path vertex, this can be very slow with a massive amount of incoming data. Furthermore, because of the way MLT works, the light paths generated in a certain time period tend to be centered around a single scene region, and if we need to generate paths from a different one, it may take quite some time for the algorithm to get there. This problem would occur in the same way if we visualised “live”, directly from the

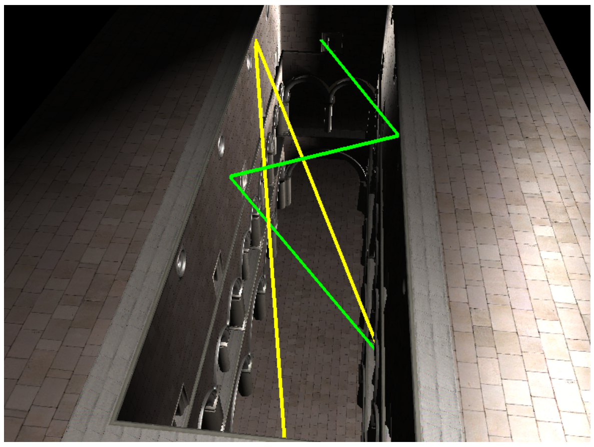
Fig. 3.3: Mutating a single path.

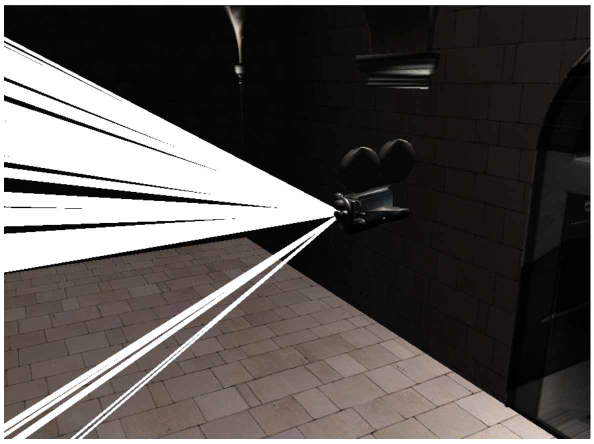
Fig. 3.4: Virtual camera.

path tracer. In fact, it would be even more severe, because path mutations can take longer the a simple file parsing.

Chapter 4

Conclusion and future work

We have presented the Metropolis Light Transport algorithm, a Monte Carlo ray tracing method for unbiased rendering, used primarily for scenes with complicated lightning. We've also shown the various methods that improved on it over the years. A library for interactive visualisation was presented, with methods for general visualisation of light paths, and also methods specifically designed with MLT in mind. A simple demo program followed, visualising data generated by Mitsuba renderer. We hope that the library will be useful in the future, either in development of a new path tracer, or an improvement of an old one, or simply for educational purposes. Possible improvements include more user-friendly interface, or additional filtering methods.

Chapter 5

Manual - Lvis key bindings

`Left mouse button` – select path or selector

`Right mouse button` – rotate camera

`F` – switch between free camera mode

`Up arrow` – move camera forward

`Down arrow` – move camera backward

`Right arrow` – camera strafe right

`Left arrow` – camera strafe left

`W` – move camera up

`S` – move camera down

`A` – switch between moving and scaling of selectors

`Z` – add green selector (start of path)

`X` – add red selector (end of path)

`C` – add white selector (any vertex, AND)

`V` – add yellow selector (any vertex, OR)

`Enter` – select active path

`Backspace` – clear all paths

`Delete` – delete active selector

`Home` – delete all selectors

`F1` – decrease the value of `limit` variable by `10`

`F2` – increase the value of `limit` variable by `10`

`F3` – decrease the value of `drawLimit` variable by `10`

`F4` – increase the value of `drawLimit` variable by `10`

`F5` – decrease the value of `threshold` variable by `10`

`F6` – increase the value of `threshold` variable by `10`

`F8` – toggle `flagSingle`

`F9` – toggle `flagPause`

`F10` – toggle `flagWaiting`

`F11` – filter drawn paths

`F12` – switch shader

References

[Bub] Michal Bubnár. Color picking tutorial. `http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=21`.

[KSK01] Csaba Kelemen and László Szirmay-Kalos. Simple and robust mutation strategy for metropolis light transport algorithm. Technical Report TR-186-2-01-18, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, July 2001. human contact: technical-report@cg.tuwien.ac.at.

[LKL$^+$13] Jaakko Lehtinen, Tero Karras, Samuli Laine, Miika Aittala, Frédo Durand, and Timo Aila. Gradient-domain metropolis light transport. *ACM Trans. Graph.*, 32(4):95:1–95:12, July 2013.

[OGL] Opengl glsl tutorial 6 - assimp 3d model loader. `http://www.youtube.com/watch?v=ClqnhYAYtcY`.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[PKK00] Mark Pauly, Thomas Kollig, and Alexander Keller. Metropolis light transport for participating media. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 11–22, London, UK, UK, 2000. Springer-Verlag.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.