BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

RENDERING OF NONPLANAR MIRRORS

ZOBRAZOVÁNÍ POKŘIVENÝCH ZRCADEL

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MILOSLAV ČÍŽ**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání diplomové práce/17981/2016/xcizmi00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel:	**Číž Miloslav, Bc.**
Obor:	Počítačová grafika a multimédia
Téma:	**Zobrazování pokřivených zrcadel**
	Rendering of Nonplanar Mirros
Kategorie:	Počítačová grafika

Pokyny:

1. Nastudujte techniky pro zobrazování pokřivených zrcadel.
2. Navrhněte metodu zobrazování pokřivených zrcadel v reálném čase.
3. Implementujte navrženou metodu.
4. Vytvořte alespoň 3 různé testovací scénáře (scény), na kterých budete demonstrovat implementovanou metodu (využijte standardní modely jako jsou: Sponza, Sibenik, San Miguel, ...).
5. Proměřte kvalitativně a výkonnostně vůči raytracingu na různých platformách (alespoň jedno gpu od NVidia a AMD). Změřte jednotlivé části metody a jak reagují na nastavení kamery, scény a materiálů. Popište podmínky, kdy je metoda kvalitativně srovnatelná s raytracingem.
6. Vytvořte video s demonstrací implementované metody.

Literatura:

- dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí:	**Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání:	1. listopadu 2016
Datum odevzdání:	24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

This work deals with the problem of accurately rendering mirror reflections on curved surfaces in real-time. While planar mirrors do not pose a problem in this area, non-planar surfaces are nowadays rendered mostly using environment mapping, which is a method of approximating the reflections well enough for the human eye. However, this approach may not be suitable for applications such as CAD systems. Accurate mirror reflections can be rendered with ray tracing methods, but not in real-time and therefore without offering interactivity. This work examines existing approaches to the problem and proposes a new algorithm for computing accurate mirror reflections in real-time using accelerated searching for intersections with depth profile stored in cubemap textures. This algorithm has been implemented using OpenGL and tested on different platforms.

Abstrakt

Tato práce se zabývá problémem přesného zobrazování zrcadlových odrazů na zakřiveném povrchu v reálném čase. Zatímco planární zrcadla nepředstavují v tomto ohledu problém, zakřivené povrchy se v dnešní době zobrazují především metodou environment mapping, která aproximuje reálné odrazy a nabízí výsledky uspokojivé pro lidské oko. Tento přístup však nemusí být vhodný např. v oblasti CAD systémů. Přesných zrcadlových odrazů se dá dosáhnout pomocí metod sledování paprsku, avšak ne v reálném čase a tudíž bez možnosti interaktivity. Tato práce zkoumá existující přístupy k tomuto problému a navrhuje nový algoritmus výpočtu přesných odrazů v reálném čase pomocí akcelerovaného hledání průsečíků s hloubkovým profilem uloženým v cubemap texturách. Tento algoritmus je implementován pomocí technologie OpenGL a otestován na různých platformách.

Keywords

mirrors, reflections, real-time, OpenGL

Klíčová slova

zrcadla, odrazy, real-time, OpenGL

Reference

ČÍŽ, Miloslav. *Rendering of Nonplanar Mirrors.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet

Rendering of Nonplanar Mirrors

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Tomáš Milet. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Miloslav Číž
May 21, 2017

Acknowledgements

I thank my supervisor, Ing. Milet, for the original idea for the work, his patience and help, and my family for their great support.

Contents

Chapter 1

Introduction

If we pay close attention to the progress being made in computer graphics, we'll find that, aside from progressively better hardware, most of it is achieved by new algorithms that use computationally cheap approximations that look realistic enough to human eye, especially in real-time interactive graphics.

Image-order rendering methods, such as ray tracing, accurately model many physical phenomena, but for a high computational cost, making them unusable in widely used real-time applications. Many researchers put effort to designing hardware and software that would allow ray tracing to be used in real-time [13]. Nevertheless, rasterization is still the absolutely prevailing method in real-time rendering engines today, and probably will be for at minimum a few more years.

Many problems of rasterization approach are caused by the light interacting with multiple object in the scene. This includes (soft) shadows, indirect illumination, refractions, caustics and, last but not least, mirror reflections. Algorithms exist for rendering of some of the mentioned phenomena, more or less accurately, but no single algorithm can yet solve the problem of global illumination as a whole, due to the nature of rasterization that treats objects in the scene separately. It is therefore necessary to address each phenomenon individually.

In case of mirror reflections, planar mirrors are easily rendered with a simple two-pass algorithm, because the scene is not distorted. Non-planar mirrors, however, pose a bigger problem. Environment mapping is the most widely used algorithm for rendering non-planar mirrors [22]. It approximates mirror reflections with significantly simplified model of the scene, stored in a texture, and generally provides a good level of realism.

Though this approach of approximating reflections in favour of performance is very suitable for certain applications, such as games, it may not be fitting the needs for accuracy of other applications, such as CAD systems. For example, a vehicle designer may find themselves in a situation when they need to check whether the driver can be blinded by reflecting light focused to their eyes by a reflecting surface.

The main goal of this work is to study the current state of rendering non-planar mirrors, experiment with new ideas and try to design a new algorithm that would allow accurate reflection rendering in real-time. The algorithm was to be implemented with OpenGL API and its performance was to be measured.

1.1 Terminology Used in This Work

This work will mostly use terminology of OpenGL [18] and cited articles. Some additional important terms include:

- cubemap object (CMO) – This term is not to be confused with cubemap texture. A cubemap object is a structure used by cubemap tracing algorithm and consists of multiple cubemap textures and other attributes.
- cubemap tracing – This name will be used for the new proposed algorithm.
- reflector – This is the object representing the mirror.
- self-reflections – This is a phenomenon when a ray is reflected off of a mirror surface more than once.
- frustum – Conventionally a view frustum is a portion of a pyramid culled by near and far planes. In context of cubemaps we'll also extend this term to include the whole pyramid, not culled by additional planes.

Chapter 2

Existing Approaches to Mirror Rendering

The area of realistic reflection rendering is not a new topic and there has been a long lasting effort that gave life to a number of methods. The following sections summarizes these methods. Chapter 5 compares them to the new method presented in this work.

It is worth mentioning that reflections are often studied together with refractions [9] because both of these fenomena involve light ray redirection off of an object's surface. Some of the following methods may therefore also be used for refraction rendering. Note that for refractions a single light ray in the scene very often refracts at least twice (when entering the object volume and when leaving it), which is analogous to self-reflections, which are typically difficult to achieve.

2.1 Rendering Planar Mirrors

As it's been stated, planar mirrors are relatively easy to be quickly rendered on modern GPUs because the scene seen in the mirror is not distorted by the mirror, only transformed by means of reflection, which is an affine transformation and can be achieved with matrix multiplication [15]. A common way to render the scene with the mirror, such as the one in fig. 2.1, is to use stencil buffer in the following way [11]:

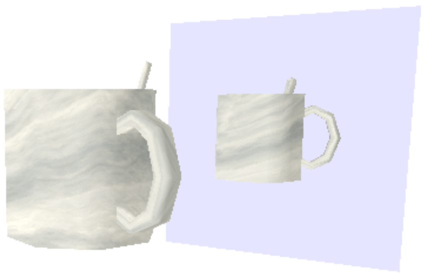


Figure 2.1: planar mirror

1. Clear stencil buffer to 0s and render the scene without the mirror.
2. Render the mirror into stencil buffer (writing 1s) and also clear depth buffer at mirror pixels.
3. Mirror the scene by the mirror plane (by changing the transformation matrix).
4. Render the mirrored scene (without the mirror) again and only on pixels where stencil buffer values are set to 1. Clipping by the mirror plane may be needed before rendering to avoid mirroring the objects behind the mirror to its front [14].

In the early days of limited GPU support for shader programming, some applications, such as the games Metal Gear Solid: The Twin Snakes (2004) or Conker's Bad Fur Day

(2001), went as far as to implement planar reflections by completely mirroring the reflected scene behind the reflector.

2.2 Deferred Shading

Deferred shading is a key technique in achieving high performance in applications using complex fragment shader programs, such as the cubemap tracing algorithm presented in this work. Deferred shading, though under different names, has been known since at least 1988's work of Michael Deering and his colleagues [5] [17].

Without this technique, expensively computed fragments are often discarded due to not passing depth test. Deferred shading aims to invoke compute-intensive fragment shaders only on fragments that actually get to be seen in the final image. This is achieved with two-pass rendering:

1. In the first pass, the scene is rendered from the camera point-of-view into so called G-buffer (geometry buffer). G-buffer contains multiple textures, each one storing some kind of data (such as a normal or world-space position) needed by the expensive fragment shader invocation. G-buffer can be constructed relatively quickly using a simple fragment shader and MRT (multiple render targets).

2. Second pass runs the expensive fragment shader on each pixel of the screen (by rendering a full-screen quad) and operates only on previously constructed G-buffer values without discarding any fragments.

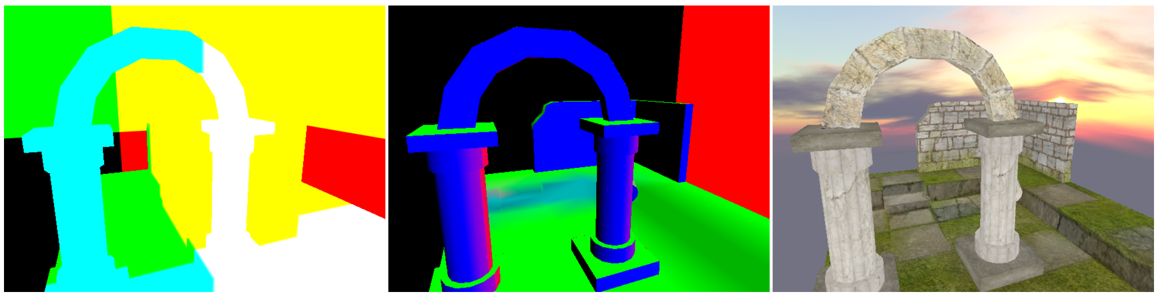


Figure 2.2: Deferred shading shown here uses world position (left), normal (middle) and texture color (not shown) for the final rendering (right).

2.3 Rendering Non-Planar Mirrors

This section explains common approaches to real-time rendering of non-planar mirrors, which is the main focus of this work.

2.3.1 Environment Mapping

One of the most widely used methods of approximating reflections on curved surfaces is so called environment mapping (or reflection mapping), introduced in 1976 [2].

Environment mapping usually offers a good balance between accuracy and speed. It works by pre-rendering the full spherical view of the scene from a certain point (usually the center point of the reflecting object) using a map projection to store the rendered view into a two-dimensional texture and later using the texture to look up pixels by rays reflected off of the mirror surface. Some of the map projections used with this method are:

- Sphere mapping – This is one of the earliest used mappings [10]. It uses a single texture to store the scene projection. The texture is constructed by rendering the scene as seen by looking (through orthographic lens) at a reflecting sphere placed in the scene at the projection center point. The technique suffers from drastic undersampling of certain parts of the scene – we say such mapping is *view-dependent* because it cannot be used for any position of the viewer without this undersampling becoming very apparent. The mapping also isn't surjective (meaning many pixels of the texture remain unused).
- Dual-paraboloid mapping – This technique was introduced in 1998 to improve on sphere mapping by using two paraboloids (and two textures) instead of a sphere [10]. This mapping samples the world more evenly and so we can call it *view-independent*.
- Cube mapping – In 1986 a way of projecting the world onto six sides of a cube has been introduced by Ned Greene. This collection of six textures is called a *cubemap*. Cube mapping offers an advantage over above mentioned methods by not applying a non-linear transformation to the scene, which can result in artifacts due to only linear interpolation support in the hardware. The technique, however, requires rendering of the scene six times and then performing several conditional jumps to decide in which of the six textures the texel is to be looked up, therefore cube mapping wasn't being widely used in the early years of OpenGL. Nowadays it is the most commonly used technique, even though it still samples the spherical surface slightly unevenly [20][10][7].
- HEALPix (Hierarchical Equal Area isoLatitude Pixelisation) – This algorithm samples the spherical surface uniformly, with each pixel having the same solid angle. The pixel lookup is expensive and MIP map filtering is complicated [20].

Other methods, such as Tetrahedron or Unicube mapping, exist, but there generally isn't one best solution to use.

Environment mapping in presented form also isn't able to model self-reflections. Dynamically changing scenes require periodical re-rendering of the environment map on the fly.

Fig. 2.3 shows the principle as well as the shortcoming of this method, i.e. the source of inaccuracy and lack of parallax. Fig. 2.4 shows the results of the method.

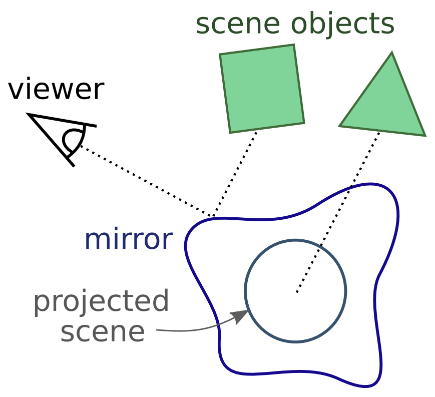


Figure 2.3: Environment mapping looks up pixels by casting a ray from the center point of the world projection in the same direction as the reflected ray.

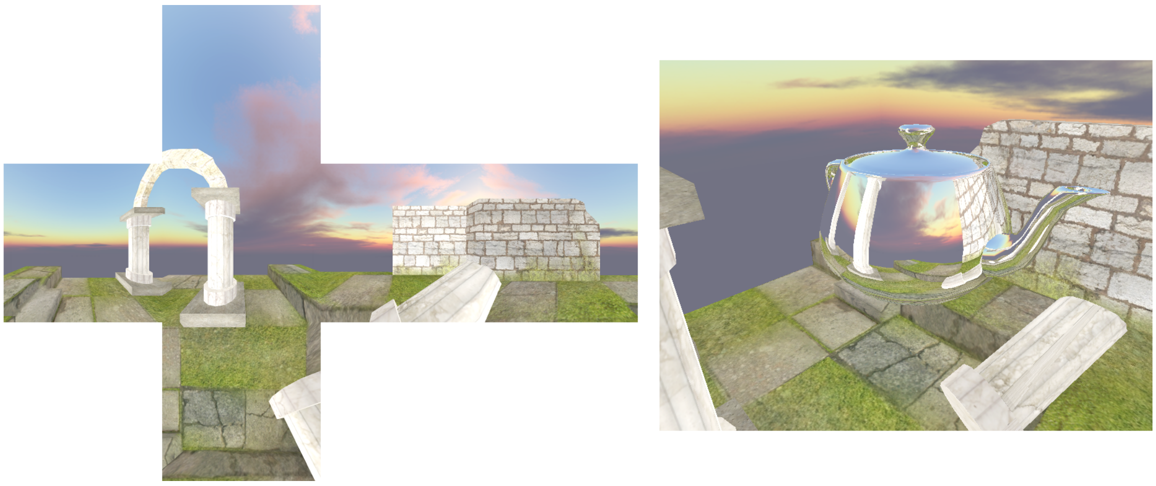


Figure 2.4: environment mapping example: cubemap projection on the left and the result scene with teapot mirror on the right

2.3.2 Parameterized Environment Maps

The disadvantage of environment maps capturing the scene from a single point is addressed by their parameterization. The work by Brian Cabral et al. [3] allows cubemaps (or other environment maps) to be parameterized by a position in space. This however comes at a great cost of having to precompute the scene at each point of parameter space with ray tracing and then compressing all the resulting textures into a convenient representation. At runtime, the environment map closest to the viewpoint is chosen and used, with possible blending for smoother results. This restricts the viewer's movement and is therefore suitable only for specialized applications.

2.3.3 Ray Tracing

Ray tracing and other similar image-order methods support accurate mirror reflections by default, but they are still unusable in real-time. Efforts are being made to bring ray tracing to real-time applications [13]. But to bring a completely new architecture model from research to wide use in practice will yet take a long time. We can therefore expect rasterization-based GPUs to remain the standard for a minimum of a few more years. Ray tracing can however be useful for this work to test the accuracy of our results.

One of the key issues of ray tracing is that a construction of either hierarchical bounding volume or space partitioning acceleration structures is needed to reduce the number of intersections for each cast ray to check. In case of dynamic scenes, such structure would need to be reconstructed at every frame. The type of best suitable acceleration structure is also dependent on the type of the scene. Furthermore, the update of per-frame structures depends in a similar way on the type of motion and animation of the objects in the scene [21]. This poses a problem that has to be dealt with when there is unpredictable movement in the scene. Another reason that keeps ray tracing from being used interactively is that it is mostly incompatible with feed-forward pipeline used in modern GPUs [16].

Many times, ray tracing works better with voxel data, since regular axis-aligned cubes are easier to traverse, intersect and build hierarchy from than general-shaped triangles. On the other hand, volumetric data have many disadvantages, such as difficult animation or high memory demands. For this reason, polygonal representation of geometry is prevailing in many areas. Voxelization of such data is possible but usually expensive. Algorithms for fast voxelization (in order of milliseconds [6]) are being developed and may become a part of the solution for real-time ray tracing, similarly to voxel cone tracing that recently made interactive indirect illumination possible [4].

2.3.4 Virtual Objects

This method was introduced in 1998 [15]. The algorithm is very similar to that of rendering a planar mirror, with one important step added – before the reflected scene is rendered, it is deformed according to the reflector surface. It can also be applied recursively to allow for mirrors being reflected in mirrors. The advantages of this method are apparent: the mirror rendering is done in a similar way to how the rest of the scene is rendered and the scene can be dynamic.

Each triangle of the reflector along with the viewer position define two spatial cells (similarly to for example shadow volumes): a *reflected cell* covering the space that is reflected in the triangle and a *hidden cell* covering the space obscured by the triangle. Only the vertices lying in reflected cells can potentially be seen in the mirror. For each of these vertices a corresponding *virtual vertex* is computed by using a relative position within the reflected cell, expressed with a triplet of barycentric coordinates. Virtual vertices are then connected to form *virtual polygons* that will form the *virtual object* seen in the mirror.

The key part is searching for the cell a given vertex falls into. This can easily be done for simple shapes such as cylinders or spheres. Acceleration structures (such as BSP trees) can be used with general shapes. Authors themselves introduce an acceleration structure called an *explosion map*, which projects the cells onto a sphere surface. Explosion map has to be recomputed each time the viewer position changes.

Challenges also arise from the fact that vertices of one polygon may lie partially in reflected/hidden/neither cells, for example in case of an reflected object intersecting the reflector. This poses a requirement to compute virtual vertices for each vertex and additionally, like in the case of planar reflections, clipping the transformed scene by the reflection surface, which in this case requires an additional z-buffer.

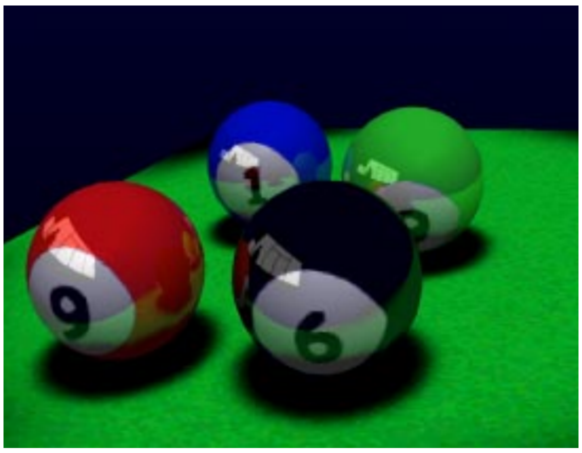


Figure 2.5: virtual objects, image from the original work [15]

The authors deal with reflections on general-shaped surface by dividing it to subparts of which each falls into one of the following categories:

- planar – This case poses no problems, as explained in section 2.1.
- convex – Reflected (hidden) cells never intersect other reflected (hidden) cells.

- concave – Reflected cells can intersect each other and therefore a vertex can lie in multiple cells simultaneously, which may be an issue. However, this case is treated in the same way as the convex case in the original work, because the problematic space areas are relatively small and the reflections look chaotic even in reality.

The issues of this method are mainly:

- It deforms the scene at vertex level, which can cause artifacts to appear if the reflected object is composed of a low number of triangles. This can be addressed by tessellating the object.
- Very complex surfaces have to be decomposed to a high number of subparts, each requiring an extra rendering pass. Such surfaces can also produce subparts that are too fine, for example if saddles are present. Manual decomposition is recommended for such cases.
- As mentioned above, concave reflections can be inaccurate. Also the explosion map has to be recomputed whenever viewer position changes.

2.3.5 Light Fields

Light field is an extension of texture, in which each texel is indexed, along with conventional u and v coordinates, with two additional coordinates s and t. These coordinates together define a ray coming from uv coordinates into the scene, so that the light field captures the scene from all possible angles within a certain range. We distinguish between light field slabs, which are planar light fields, and surface light fields, that capture the rays from each point of arbitrary surface.

Light field slabs were used in a work by Wolfgang Heidrich et al. in 1999 [9] to store a ray database that could be used to render accurate reflections and refractions. In 2005, a work titled Real-Time Reflection Mapping with Parallax [22] built on this method to form a very promising algorithm.

Six precomputed light fields are arranged as a cube around the reflector. This light field cube is then queried with reflected rays to find the reflected color (fig. 2.6). Though the algorithm performs very well in many situations, it has the following disadvantages:

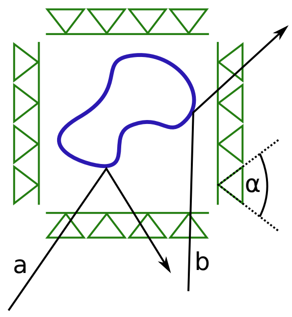


Figure 2.6: Reflector is surrounded by a cube of light fields (green) parameterized by the angle α which are queried by reflected rays. Note that ray b cannot yield a result as the angle it enters into the light field is too sharp.

- Light fields require a lot of memory to store and their construction is expensive.
- The algorithm doesn't model self-reflections.
- Rays coming under very sharp angles against the light field cannot be looked up in the basic version of the algorithm. The authors show how more light fields can be utilized in order to guarantee any ray intersecting the light field cube can be looked up, but this costs more resources.

- Rays that miss the light fields cannot be looked up, which can happen if the reflecting object moves or deforms outside of the boundaries of the enclosing light fields.

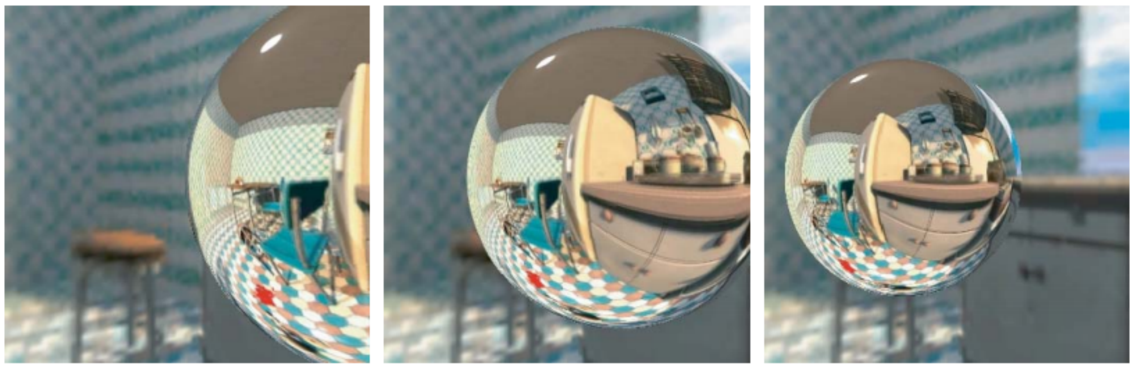


Figure 2.7: Real-Time Reflection Mapping with Parallax, image from the original work [22]

2.3.6 Reflection Space Image Based Rendering

Alternative approaches exist, such as image-based methods [3]. In the cited work, sphere maps are used to capture the scene from multiple different points. This information is then used to create an image from arbitrary viewpoint positions with the help of warping. The idea of capturing the scene at multiple points into environment maps is reused in this work.

2.3.7 Sample-Based Cameras

Voicu Popescu et al. [16] presented quite complex method that nevertheless achieves good results, including self-reflections. It is based on constructing helper pinhole cameras whose views form the virtual image seen in the mirror. The cameras are stored in BSP trees. Aside from the ray map, camera and BSP tree constructions and tessellation of reflected geometry, it is also required (in a manner similar to virtual objects) that complex reflector geometry is divided to so called simple and complex parts, of which the complex ones are rendered using environment mapping.

Chapter 3

New Cubemap Tracing Algorithm

The following sections describe the principle of the new algorithm, the main contribution of this work.

3.1 Cubemap Properties

As this work relies on cube mapping, we'll further explore the cubemap projection properties so that they can be used in later chapters.

Definition 3.1. Projection of point $\vec{p}$ onto cube C is an intersection $\vec{q}$ of the cube C surface and a semi-straight line coming from the center point of C towards the point $\vec{p}$.

Theorem 3.2. *All points of a line segment in 3D space projected onto a cube lie on a single plane.*

Proof. Let $\vec{c}$ be the cubemap center point and l_1 the line segment being projected, defined by two points p_1 and p_2 as

$$\vec{l_1}(t) = t \cdot \vec{p_1} + (1 - t) \cdot \vec{p_2}, \quad t \in [0, 1]. \tag{3.1}$$

Let's suppose l_1 doesn't intersect the cube (if it does, we can simply consider a smaller cube with the same center point). For each t, the point $\vec{l_1}(t)$ is by definition 3.1 projected to a point that has to lie on another line segment l_2 defined by the point $\vec{l_1}(t)$ and the cubemap center $\vec{c}$ as

$$\vec{l_2}(s, t) = s \cdot \vec{l_1}(t) + (1 - s) \cdot \vec{c}, \quad s \in [0, 1]. \tag{3.2}$$

By substituting 3.1 to 3.2 and simplifying we get

$$\vec{l_2}(s, t) = s \cdot (\vec{p_2} - \vec{c}) + st \cdot (\vec{p_1} - \vec{p_2}) + \vec{c} \tag{3.3}$$

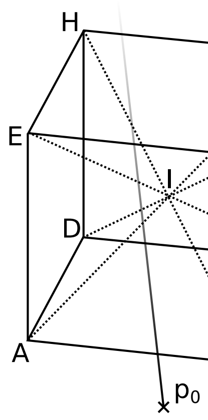
which is a parametric form of an equation that defines a subset of a plane as a linear combination of two direction vectors ($\vec{p_2} - \vec{c}$ and $\vec{p_1} - \vec{p_2}$) plus a constant offset vector ($\vec{c}$). $\square$

This also means that the projection to each side of the cube is rectilinear [12], i.e. straight lines in 3D space are projected to straight lines at the projection plane. This is obvious as the intersection of the plane $\vec{c}\vec{p_1}\vec{p_2}$ with another plane (the cube side) is a line.

These facts are important because they allow us to easily trace a line projected onto a cube surface by projecting only two points that define the line and using a linear interpolation of the cubemap coordinates for the rest of the points, even if the line is projected onto multiple sides, as seen for example in fig. 3.5.

Theorem 3.3. *Any semi-straight line can be projected to at most four sides of a cube, unless it intersects its center point.*

Proof. 1. Consider a cube with vertices named as in the picture:



Furthermore let's have an arbitrary semi-straight line l that doesn't intersect I and starts from a point p_0. Let each side of the cube define an infinite view frustum by all points in space that are projected to it. The whole space is then subdivided into six such frusta, one for each side (eg. $EFGHI$). It is obvious that in order for l to be projected to given side, it has to intersect its frustum. Therefore we want to show that l can intersect at most four frusta.

2. p_0 has to lie in exactly one frustum. Since the cube is symmetric, let us without the loss of generality suppose it lies in frustum defined by $ABCDI$ (bottom).

3. Tracing l from p_0, it now has to enter one of the neighboring frustums: $ABEFI$, $BCFGI$, $CDGHI$ or $ADEHI$. The cases are symmetric so suppose it enters $ABEFI$ (front).

4. By entering the frustum, l intersected the ABI plane and can no longer enter any frusta whose whole volume lies below the plane. These are $ABCDI$ (the starting frustum) and $CDGHI$ (back).

5. Going further along l, it can now enter one of the following frusta:

   (a) $ADEHI$ – In the same way we now eliminate $ABEFI$ and $BCFGI$. The only unvisited uneliminated frustum is now $EFGHI$ (top), by entering which $ADEHI$ gets eliminated, leaving no frustum left. Three inter-frustum transitions were made in total, i.e. four frusta are intersected.

   (b) $BCFGI$ – Symmetric to previous case, we get to the same result.

   (c) $EFGHI$ – $ABEFI$ and $ABCDI$ get eliminated. The only unvisited uneliminated frusta are now $ADEHI$ (left) and $BCFGI$ (right), which are symmetric cases. Entering $ADEHI$ eliminates $BCFGI$ and $EFGHI$ and leave no frustum to go to next. Again, three transitions were made and four frusta are intersected.

□

3.2 Basic Principle

The cubemap tracing algorithm, similarly to environment mapping described in section 2.3.1, uses cubemap textures to model the surrounding world being reflected in the mirror. However, unlike environment mapping, the cubemap tracing algorithm doesn't only use one cubemap texture. Multiple cubemap textures are grouped into so called *cubemap object* (CMO). One to multiple CMOs can be used with cubemap tracing. The basic version is designed to be implemented with fragment shaders, a compute shader version will be presented later.

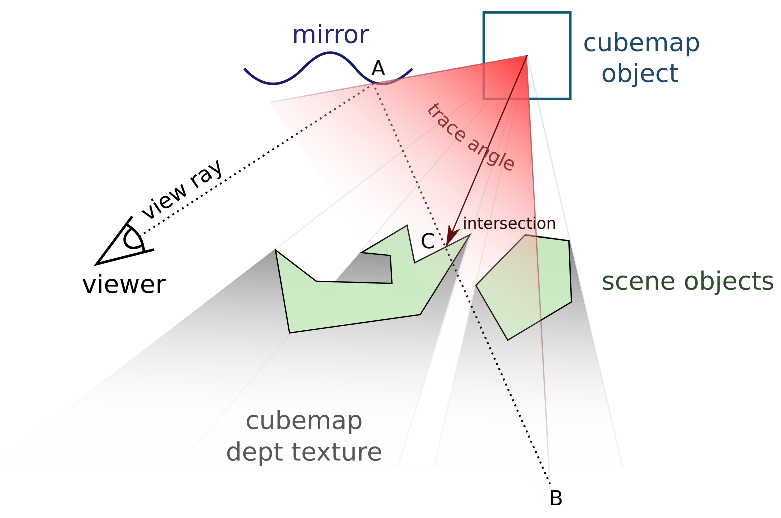


Figure 3.1: Principle of the cubemap tracing algorithm, using one CMO. The reflected view ray is iteratively traced along an angle defined by the mirror-incidence point (A) and a distant point of the ray (B). The goal is to find the intersection with scene objects (C), which is possible because of the objects being captured in the cubemap depth texture.

Suppose a set of N CMOs $S = \{\Theta_i \mid i \in \{1 \dots N\}\}$. Each CMO Θ_i has the following attributes:

- a world position $\Theta_i^{\vec{p}}$, not necessarily inside the mirror,
- diffuse color cubemap texture Θ_i^C,
- world distance cubemap texture Θ_i^D, computed from depth texture,
- mirror mask and normal cubemap textures, Θ_i^M and Θ_i^N respectively, if self-reflections are to be used,
- hierarchical cubemap acceleration texture Θ_i^A, if acceleration is to be used.

The cubemap textures, such as Θ_i^C, Θ_i^D etc., can be sampled using a 3D direction vector emanating from the cube center, as defined by OpenGL [18, p. 241]. This is denoted with brackets, e.g. $\Theta_i^D(\vec{u})$. Let us also suppose that the cubemaps are axis-aligned and have the same resolution, with the side length in pixels being an integer power of two.

Each CMO should be placed at different position in the scene, not necessarily inside the mirror.

Using distance texture instead of non-processed dept texture prevents shaders from having to recompute each depth value (which is usually in logarithmic scale) to actual world distance, and is therefore better for performance reasons.

In the very basic version of the algorithm, the Θ_i^C and Θ_i^D cubemap textures are constructed before rendering the mirror. This is done by rendering the scene without the mirror object. When rendering the mirror pixels, a reflected ray is constructed, using the viewer world position, the view sample world position and normal at the incidence point. The ray is then iteratively traced in each CMO, checking against the distance texture Θ_i^D for intersection (within given error e) in each step.

3.3 Self-Reflections

The algorithm potentially supports rendering self-reflections, although the feature can also be turned off. If we want self-reflections allowed, the algorithm is extended as follows.

The position of all CMOs should be outside the reflector, as the reflector geometry will be rendered into cubemap textures (Θ_i^M, Θ_i^N etc.). CMOs placed inside the mirror would either mostly not capture the mirror if backface culling is turned on, or see only the inside of the mirror and nothing else in the scene if backface culling is turned off. (Note that the latter might still work if there are enough CMOs outside the reflector – this is left to further research.) At render time, the rays are traced in the same way as they would normally be. If an intersection is found, it is decided, using the values in Θ_i^M mirror mask texture, whether the intersection lies at the reflector surface or not. If it does, a new reflected ray is constructed (using the position of the intersection and mirror surface normal stored in Θ_i^N) and traced again. The new ray cannot be traced from the exact start, because by definition it always starts at an intersection with the reflector, which would immediately terminate the tracing. For this reason, a bias value b is added to initialize the ray further away from its origin. This value is a constant in alg. 1, but later experiments show that it's difficult to set and should probably be computed individually for each reflected ray.

A different strategy that would avoid the need for bias value would be to search for a first intersection that comes after a non-intersection sample.

Data:

fragment shader inputs:

$\vec{p_1}$	world position of the mirror surface point being viewed
$\vec{n}$	surface normal at $\vec{p_1}$
$\vec{v}$	position of the viewer
S	the set of CMOs as described above

algorithm parameters:

e	intersection distance limit
s	interpolation step, $s \in (0, 1)$
l	traced ray length
r	upper bounce limit for self-reflections
b	self-reflection bias, $b \in (0, 1)$

Result: reflected color

```
1  $ray\_no \leftarrow 0$
2  $\vec{p_2} \leftarrow \vec{p_1} + l \cdot reflect(\vec{p_1} - \vec{v}, \vec{n})$        // get the ray end point
3
4  if $ray\_no = r$ then
5      return not found
6  end
7
8  for $\Theta \in S$ do                                         // for each cubemap
9      $t \leftarrow 0$ if $ray\_no = 0$ else $b$
10
11     while $t \leq 1$ do                                       // trace the ray
12         $\vec{p} = interpolate(\vec{p_1}, \vec{p_2}, t)$              // get the ray point
13         $\vec{u} = normalize(\vec{p} - \vec{\Theta^p})$                // get the cubemap coords
14
15         if $|len(\vec{p}, \vec{\Theta^p}) - \Theta^D(\vec{u})| \leq e$ then   // intersection?
16             if $\Theta^M(\vec{u})$ then                          // intersection on mirror?
17                 $\vec{p_1} \leftarrow \vec{\Theta^p} + \vec{u} \cdot \Theta^D(\vec{u})$   // init new position
18                 $\vec{n} \leftarrow \Theta^N(\vec{u})$                   // init new normal
19                 $ray\_no \leftarrow ray\_no + 1$
20                 go to line 2                              // trace the new ray again
21             else
22                 return $\Theta^C(\vec{u})$                       // return the color
23             end
24         end
25         $t \leftarrow t + s$
26     end
27 end
28
29 return not found                                          // no intersection found
```

Algorithm 1: Cubemap tracing algorithm with self-reflections. (Without self-reflections, lines 16 to 21 would be dropped.) Distance threshold criteria for intersection and constant-step sampling are used here.

3.4 Intersection Criteria

Unlike with ray tracing, tracing individual rays in cubemap tracing happens in iterations of discrete steps. We can use different criteria for deciding whether the ray intersects the distance profile in given iteration. Let

$$\Delta_i = \delta(p_i) - \delta(d_i)$$

where $\delta(p_i)$ is the distance of the point p_i on the ray to the cubemap center in ith iteration and $\delta(d_i)$ is the value given by the distance profile in ith iteration (see line 15 in alg. 1). We can now define two different intersection criteria:

- *distance threshold* – We declare the intersection to have happened if and only if $|\Delta_i|$ is below some predefined threshold ϵ. Let t_d^ϵ be the the first intersection found using the distance threshold criterion, in terms of the ray parameter t. Simply put, we declare an intersection when the ray gets close enough to the distance profile.
- *analytical intersection* – We keep a record of the distance in the previous iteration Δ_{i-1}. We declare the intersection if and only if the sign of Δ_i and Δ_{i-1} differ. Note that this is affected by the iteration step length l. Let t_a^l be the the first intersection found using this criterion, in terms of the ray parameter t. Simply put, we declare an intersection if the ray crosses the distance profile curve.

Possibilities arise to set up different strategies for searching for the intersection, possibly combining the above defined criteria. We may for example utilize distance threshold and search for t_d^ϵ but also keep the record of the first intersection t_a^l that we can use in case we're unable to find t_d^ϵ. We may also keep multiple analytical intersections using different l values.

Another strategy may be based on switching between distance threshold and analytical intersection depending on some parameters, for example on how close the ray is to being perpendicular to the cubemap side (which can quickly be decided with a dot product of the cubemap coordinates of the start and end point of the ray).

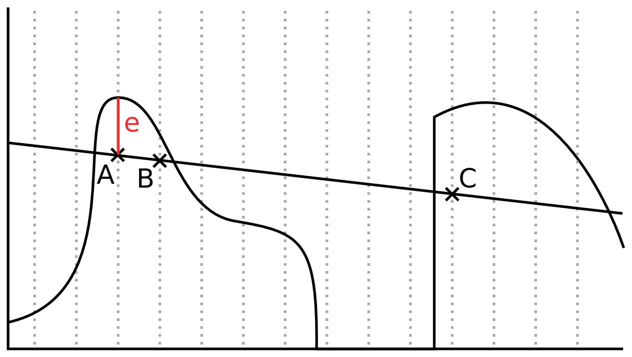


Figure 3.2: Intersection criteria: distance threshold will detect A as an intersection given high enough threshold value ($\geq e$), but such value may also detect points we wouldn't consider an intersection. Lower threshold value may miss A and detect B or even miss B and fail completely. Analytical intersection will detect A. Note that it would also detect C, which might be undesirable.

Visual differences and impacts on the number of iterations needed to trace the ray can be seen in the fig. 5.1.

3.5 Ray Sampling Strategies

By the length of the step, we'll define three sampling strategies:

- *constant angle* – This is a simple strategy that takes use of the fact (proven in section 3.1) that we can linearly interpolate between the cubemap coordinates of the projected start and end point of the ray. Each step is constant in angle. This method suffers from undesirable increasing step length along the ray and decreasing sampling density in the distance, possibly skipping valid intersections and causes distant objects to not be seen in the mirror.

- *constant step* – This strategy keeps a constant step length instead of angle by interpolating the position along the ray (not the cubemap coordinates). The sampling density is kept the same and therefore better results are obtained. However, more steps are needed and with rays close to perpendicular to the projection plane many texels can be sampled multiple times (because the step angle gets very small), which is inefficient.

- *optimal* – Our goal when tracing the projected ray is to sample each texel at most once. This might be achieved with a line rasterization algorithm. Another way is to, after sampling a texel, move to the next one using the below presented algorithm 4 – this effectively achieves rasterization, but the algorithm is not simple enough to be used very often, so this strategy is to be considered and tested.

Two of these strategies can be seen in fig. 3.3. We might also try to create strategies that sample the ray in different order, e.g. backwards, randomly etc. These strategies should nevertheless be applied carefully, as we need to find the first (i.e. closest) intersection to the incidence point.

3.6 Acceleration

Many searching algorithms can be accelerated with hierarchical acceleration structures, such as k-d trees or image pyramids, such as hierarchical z-buffer [1] [8] [19]. Our case of searching for the first intersection of a ray with arbitrary distance profile can be accelerated using an image pyramid structure. In essence, the approach is similar to minimum bounding rectangles, applied to pixel values.

The structure, in 1D, can be seen in the figure 3.4. The hierarchy consists of $[min, max]$ interval values and can be quickly computed by parallel reduction. Issues that burden many hierarchical structures, such as tree balancing, are also avoided here. We can see that at the beginning of each section of the ray it can sometimes be determined that it cannot contain the intersection with the distance profile. The section can then be skipped.

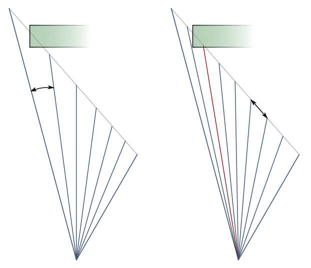


Figure 3.3: sampling strategies: constant angle (left, note the skipped intersection) vs constant distance (right)

Let us refer to the sections of the acceleration structure as *tiles*. In case of cubemap representation, i.e. multiple 2D textures addressed with a 3D vector, the world-space boundaries of a tile are not as easily determined as in 1D case, as seen in fig. 3.5. A helper subroutine is needed.

This subroutine is called `acc_bounds` in alg. 2, and its basic structure is shown in alg. 4. It takes the current ray, position on the ray and acceleration level as its input and returns the next and previous boundary in terms of interpolation parameter t.

Using this algorithm, we expect to reduce the number of iterations needed to find the intersection. The exact amount of acceleration that will be achieved in practice is however very difficult to find. This is caused by the probabilities (of intersection occurring, of being able to skip a tile etc.) depending very much on the properties of the scene. The values present in the distance texture clearly do not show any common probability distribution and are usually locally dependent on each other, i.e. values close to each other will very likely be close in value because there are a lot of flat surfaces in the scene. The worst and best case scenarios stay the same for both accelerated an unaccelerated cases, but the average case should benefit from acceleration. The question of how much exactly is for the mentioned reasons left for testing in later chapters.

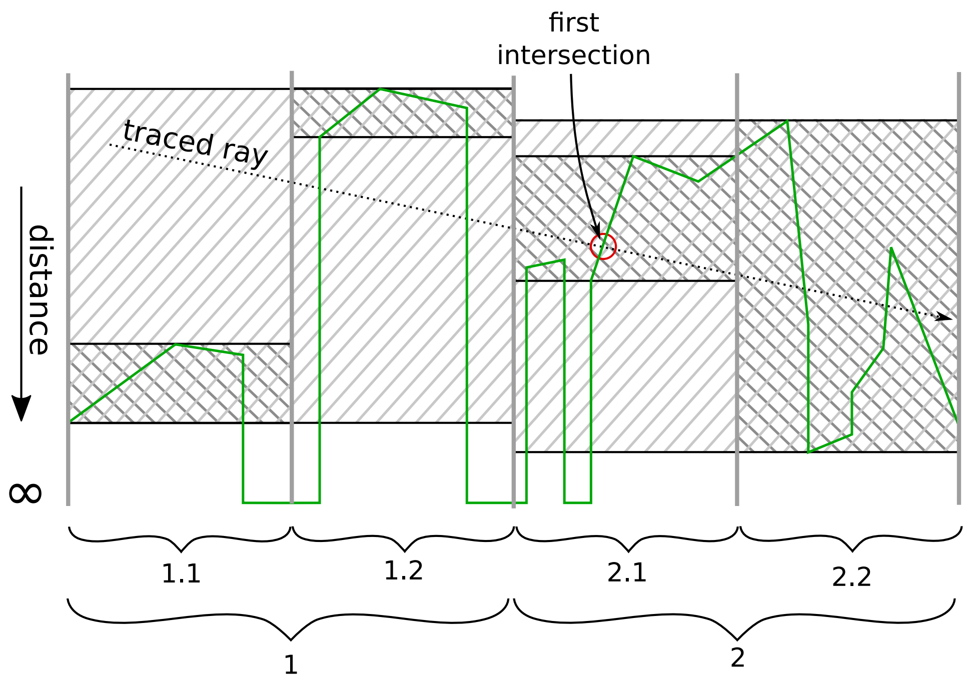


Figure 3.4: Acceleration structure used to quickly find the first intersection of the traced ray with the distance profile, simplified to 1D. In this case, two-level structure is used: first level consists of two sections (1 and 2), the second level is a subdivision of the first level and consists of four sections (1.1, 1.2, 2.1 and 2.2). Each section contains a $[min, max]$ interval of distance over the area it covers. This information can be used to quickly discard sections that cannot contain the intersection, in this case 1.1 and 1.2.

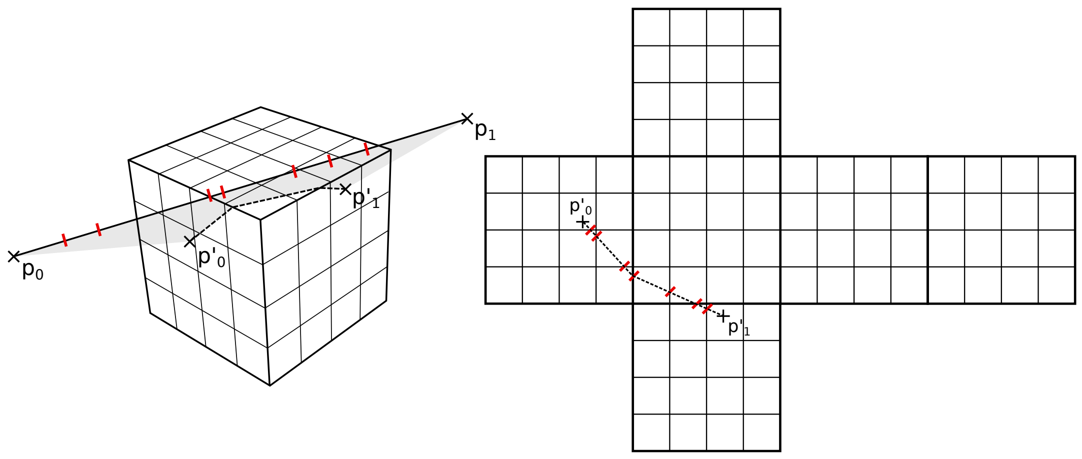


Figure 3.5: Ray projected onto cubemap acceleration structure can span over multiple cubemap sides (but no more than four, as proven in section 3.1). Red color marks the acceleration structure boundaries.

Data:

inputs:

$\Theta, t, \vec{p_1}, \vec{p_2}$	values from algorithm 1
B	helper array storing the next boundary value (in terms of t), for each acceleration level, initialized with 0s

parameters:

a_{from}	lowest acceleration level to use
a_{to}	highest acceleration level to use

```
for $i \in \{a_{from} \ldots a_{to}\}$ do
    if $t \geq B(i)$ then
        $min, max \leftarrow \Theta^A(\vec{u}, i)$                              // section dist.  interval
        $t_{next}, t_{prev} \leftarrow acc\_bounds(\vec{p_1}, \vec{p_2}, t, i)$           // section boundaries
        $d_{next} \leftarrow dist(\vec{\Theta^p}, interpolate(\vec{p_1}, \vec{p_2}, t_{next}))$   // distance at boundary
        $d_{prev} \leftarrow dist(\vec{\Theta^p}, interpolate(\vec{p_1}, \vec{p_2}, t_{prev}))$   // distance at boundary

        if $(max < d_{next} \wedge max < d_{prev}) \vee (min > d_{next} \wedge min > d_{prev})$ then
            $t \leftarrow t_{next}$                                      // skip the section
            break
        else
            $B(i) \leftarrow t_{next}$                                   // next section boundary
        end
    end
end
```

Algorithm 2: Acceleration algorithm to be inserted before the line 12 in algorithm 1. It checks if a section of ray can be skipped and if so, modifies the value of t.

3.7 Use of Compute Shaders

This section will describe an alternative version of the algorithm that uses compute shaders instead of fragment shaders.

The acceleration structure for this case is slightly different from the above presented version – the subdivision into cells alternates between 4×8 and 8×4 at each level, as shown in fig. 3.6.

The algorithm itself is presented as alg. 3. It takes direct use of the acceleration structure and so an unaccelerated version isn't presented. The algorithm traces the ray by traversing the acceleration structure depth-first for each side of the cube the ray is projected to. Maximum of four sides will be searched for a single ray, as proven in section 3.1.

To perform rendering with compute shaders, we firstly create a shader storage buffer B, which is then filled by the fragment shader with the information for the following compute shader dispatch. After the fragment shader pass, the buffer B should, for each visible mirror pixel, contain the pixel coordinates and information that defines the reflected ray to be traced for the pixel. Compute shader dispatch is then started. The size of the dispatch in x-direction is defined as the number of mirror pixels stored in B divided by 8, as each workgroup will be processing 8 pixels, and the size in other dimensions is left at 1. The local size of the workgroup is 32×8 (32 for the number of threads processing a single pixel and 8 for the number of pixels processed by a single workgroup). 32×8 makes workgroups the size of 256.

The compute shader version of the algorithm is potentially more efficient because it effectively achieves rasterization and samples each texel at most once by default.

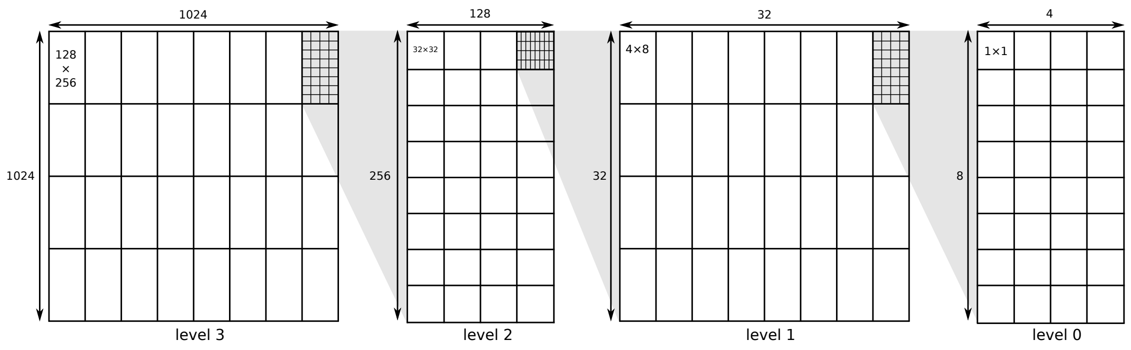
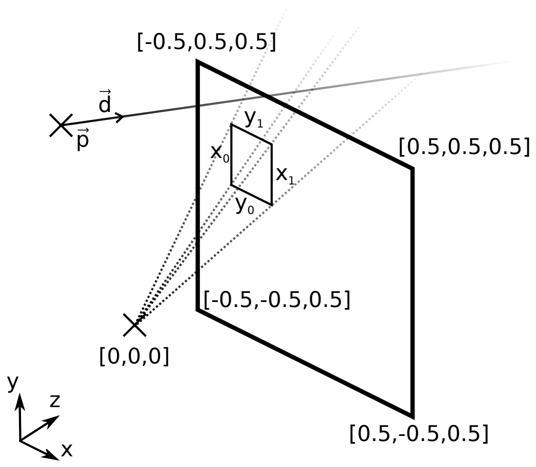


Figure 3.6: The acceleration structure used with compute shaders is subdivided into 8×4 tiles so that the number of tiles processed at the current level is always 32, which is the warp size on NVidia hardware and also the argument size of the ballot instruction, in bits. 8×4 and 4×8 subdivisions are being alternated between the levels so that the tile width to height ratio doesn't fall under 0.5 in order to keep the depth dispersion within it as low as possible. The base resolution of 1024×1024 was chosen to fit this way of subdivision. Image subdivided in this way can be stored in MIP maps, as shown in fig.4.2.

Alg. 3 requires deciding whether given ray intersects a subspace defined by the acceleration structure cell (function `intersects_cell`). This can be done as follows.

Firstly we transform the coordinate system of the current cubemap side to the coordinate system shown in the picture:



We want to decide whether a ray defined by the origin point $\vec{p}$ and a unit direction vector $\vec{d}$ as

$$\vec{p} + t \cdot \vec{d}, \quad t \in [0, \infty] \tag{3.4}$$

intersects the infinite cell frustum defined by points $[0,0,0]$, $[x_0, y_0, 0.5]$, $[x_1, y_1, 0.5]$ as all points $\vec{s}$ such that

$$\frac{\vec{s}_z}{0.5} \cdot x_0 < \vec{s}_x < \frac{\vec{s}_z}{0.5} \cdot x_1 \quad \wedge \quad \frac{\vec{s}_z}{0.5} \cdot y_0 < \vec{s}_y < \frac{\vec{s}_z}{0.5} \cdot y_1. \tag{3.5}$$

By substituting 3.4 to 3.5 we get a system of linear inequalities, which we are able to simplify to

$$\begin{aligned} 2x_0\vec{p}_z - \vec{p}_x < t(\vec{d}_x - 2\vec{d}_z x_0) \quad &\wedge \\ 2y_0\vec{p}_z - \vec{p}_y < t(\vec{d}_y - 2\vec{d}_z x_0) \quad &\wedge \\ \vec{p}_x - 2x_1\vec{p}_z < t(2\vec{d}_z x_1 - \vec{d}_x) \quad &\wedge \\ \vec{p}_y - 2y_1\vec{p}_z < t(2\vec{d}_z y_1 - \vec{d}_y). & \end{aligned}$$

Solving for t we get the final system

$$t \overset{?}{\neq} \frac{2x_0\vec{p}_z - \vec{p}_x}{\vec{d}_x - 2\vec{d}_z x_0} \quad \wedge \quad t \overset{?}{\neq} \frac{2y_0\vec{p}_z - \vec{p}_y}{\vec{d}_y - 2\vec{d}_z x_0} \quad \wedge \quad t \overset{?}{\neq} \frac{\vec{p}_x - 2x_1\vec{p}_z}{2\vec{d}_z x_1 - \vec{d}_x} \quad \wedge \quad t \overset{?}{\neq} \frac{\vec{p}_y - 2y_1\vec{p}_z}{2\vec{d}_z y_1 - \vec{d}_y}$$

where each $\overset{?}{\neq}$ inequality symbol is either $>$ if the fraction denominator is positive or $<$ if it is negative (zero means a boundary case). We want to find whether the system has a solution, which can very easily be done by creating an interval of solutions $[t_0, t_1]$ and checking whether $t_1 \geq t_0$, in which case the line intersects the cell space (otherwise not).

If there exists a solution, we should additionally check for its validity. Particularly, negative values of t (solutions that do not lie on the specified semi-straight line) and solutions with negative z coordinate (which may satisfy the equation 3.4 but do not lie in the „forward“ spanning frustum) should be considered invalid. We can do this by substituting t_0 and t_1 values into the ray equation by which we get two solutions and we check whether at least one of them satisfies the conditions.

The function `intersected_side_frusta` returns ordered list of sides to which given ray will be projected. The list can be constructed as follows:

1. Decide the first side by taking the ray starting point and finding the largest magnitude of its coordinates [18, p. 241].

2. Decide the last side the same way with the ray end point.

3. Decide the remaining sides between the first one and the last one. There can be at most two sides left, as shown in section 3.1. This can be done by checking all remaining frusta with the function `intersects_cell` that was described in previous paragraphs. (The order of these two sides can be decided by calculating the t parameter.)

Alternatively, all side frusta can just be checked as in point 3. The sides may also be left unordered, in which case all of them have to be searched and of all intersection found, the closest one to the cube center will be returned.

The function `first` takes the map of the current acceleration level and returns the first cell that should be checked for intersection, i.e. the first cell the ray intersects. We need the first cell because we are looking for the first intersection along the ray. The implementation of the function can be simple: at the start of the invocation the direction of the projected ray in the cubemap side coordinates is computed. This can be one of four values: RB (left-to-right, top-to-bottom), LB (right-to-left, top-to-bottom), RT (left-to-right, bottom-to-top), LT (right-to-left, bottom-to-top). The cell mask will then be traversed line by line in the same direction and the first cell will be returned (fig. 3.7).

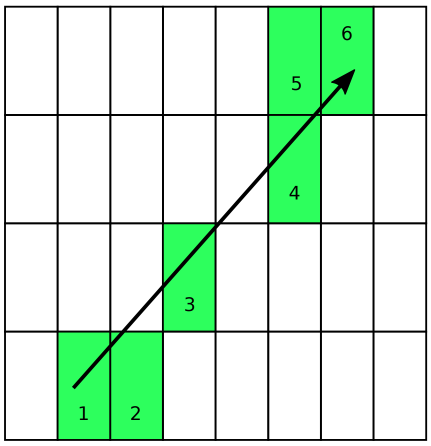


Figure 3.7: At each level of the acceleration structure a binary mask is constructed. The mask says which cells may contain the intersection (this is a subset of cells that the ray intersects, or, in other words, gets rasterized to) – an example is shown in the figure. The cells should proceed to be checked in the order the ray intersects them as shown in the figure. The order is decided by going through the cells in the same direction as the projected ray (left-to-right, bottom-to-top in the picture).

Data:

inputs:

S	the set of CMOs as described in the section 3.2
$\vec{p_1}, \vec{p_2}$	the traced ray start and end point, respectively
n	ID of the shader invocation, $n \in \{0 \dots 31\}$
$levels$	number of levels of the acceleration structure

shared between threads:

m	cell map of boolean values, indexed by level and shader ID

Result: reflected color

```
1  for $\Theta \in S$ do                          // for each cubemap
2      $sides \leftarrow intersected\_side\_frusta(\vec{p_1}, \vec{p_2}, \vec{\Theta^p})$
3      for $side \in sides$ do
4          $level \leftarrow 0$
5          $cell \leftarrow (0, 0)$               // cell being checked by the workgroup
6          $backtracking \leftarrow false$        // whether coming from top or bottom
7          loop                                   // examine the next cell
8              if $not\ backtracking$ then
9                  $m(level, n) \leftarrow intersects\_cell(\vec{p_1}, \vec{p_2}, \vec{\Theta^p}, level, n) \wedge$
                     $intersection\_possible(\vec{p_1}, \vec{p_2}, level, n)$
10                 wait on barrier                // ballot instruction in GLSL
11             end
12             $cell \leftarrow first(m, \vec{p_1}, \vec{p_2})$      // first intersected cell
13             if $cell\ is\ none$ then           // no more cells to check?
14                 $level \leftarrow level - 1$   // go up
15                 $backtracking \leftarrow true$
16                 if $level < 0$ then
17                     break                      // no intersection at this side
18                 end
19             else
20                 $m(level, n) \leftarrow false$ // mark the cell checked
21                 $level \leftarrow level + 1$   // go down
22                 $backtracking \leftarrow false$
23                 if $level = levels$ then
24                     return $get\_pixel(\Theta, side, cell)$    // intersection found
25                 end
26         endloop
27     end
28 end
29 return $not\ found$
```

Algorithm 3: Compute shader version of cubemap tracing algorithm executed by 32 threads for a ray corresponding to a single pixel – we start at the top subdivision level (0), each invocation checks for intersection with one of the 32 cells and broadcasts the binary information (the cell mask) to others. Once the complete 32-bit information is available to all invocations, they find the first intersected cell in the direction of the ray and proceed to check it the same way at a lower level.

3.8 Number and Placement of CMOs

Only those parts of the scene can be reflected in the mirror that are seen from one of the CMOs' positions, which also affect the resolution the different parts of the scene are captured in. We should therefore examine ways the CMOs should be placed in the scene.

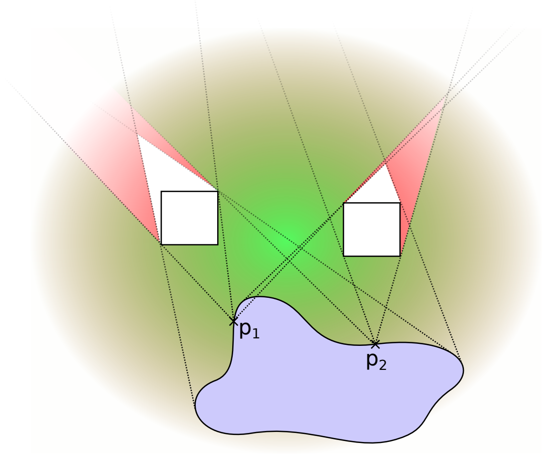


Figure 3.8: Potentially reflected area (red) can be approximated (green) by placing the cubemaps on the reflector surface (points p_1 and p_2).

Similarly to work of Ofek and Rappoport [15], let us define a *potentially reflected area* of the scene as a set of all positions from which at least one point on the reflector surface can be seen – or vice versa, by the laws of geometrical optics, the set of all points that can be seen from at least one point on the reflector surface. Let the complementary subspace be called *hidden*.

Let P be the set of all points on the reflector surface. Given a single point $p_a \in P$, let $V(p_a)$ be the set of all scene points directly visible from position p_a. We can obtain the potentially reflected area as an union $\bigcup_{p \in P} V(p)$. Though this is an union of infinitely many sets, we intuitively see that usually the first few points we add contribute the vast majority of visible points. We can therefore very well approximate the whole potentially reflected area by constructing the union for only a few points that are separated by considerable distance at the surface. We can use those points as the positions for the CMOs, i.e. we place the CMOs directly at the reflector surface to approximate the potentially reflected area.

Note that in case of self-reflections when the reflector itself has to be seen by the CMOs, we should leave an offset between the CMO and the surface in order to prevent projecting the intersected reflector surface as a perpendicular area, clipping the surface with near plane during rendering to the cubemap etc.

The number of CMOs used may depend on the complexity of the scene, but shouldn't be too high, as each CMO adds to the demands on computational time and will contribute only little to the total potentially reflected area covered, as mentioned in previous paragraphs. On the other hand, if the intersection is found early, the remaining CMOs don't have to be searched. Attention should therefore be payed to the order in which CMOs are placed – place the first one to cover the most likely parts of the scene to be reflected, similarly the second one etc. Note that the order can be changed without having to recompute the CMOs, which might be a basis for optimization.

Another variable that may be considered is the viewer position – depending on the application we may be able to predict it beforehand, or we may periodically, or with any significant change of view position, rearrange the CMO setup accordingly to reflect the current viewer position. The question of how exactly to adjust the setup remains to be researched. One idea might be to quickly estimate, on a large scale, which parts of the scene are potentially reflected given the current viewer position and try to cover these with the cubemap view as much as possible. Another, probably more expensive but also more straight-forward approach, would be to try out a few random CMO setups and choose the one that produces the least amount of unresolved pixels.

Note that changing the position of CMO requires its textures to be re-rendered and acceleration structures to be recomputed. This is however not very expensive, as is the case for example with light fields, and can be done on the fly.

For now it is advised to place the CMOs manually or randomly with the following heuristics in mind:

- The position should be close to the reflector surface (not directly at it with self-reflections enabled).
- The positions of different CMOs should be as far as possible from each other.
- The CMOs should be ordered by how likely they will successfully yield an intersection.

3.9 Dealing with Unresolved Intersections

The algorithm may be unable to find the intersection. Depending on demands of the application, we may choose to address this issue in one of the following ways:

- Marking the pixels as unresolved, for example with a specific color.
- Utilizing multiple intersection criteria, described in section 3.4, to maximize the probability of finding an intersection.
- Using the best candidate found, i.e. the sample closest to being intersection.
- Filling the unresolved pixels with approximations, for example using traditional environment mapping (fig. 3.10), interpolating between neighbouring pixels etc.
- Filling with background (skybox texture, ...).
- Applying a different, possibly computationally more intensive method, such as ray tracing, to fill in the missing information.

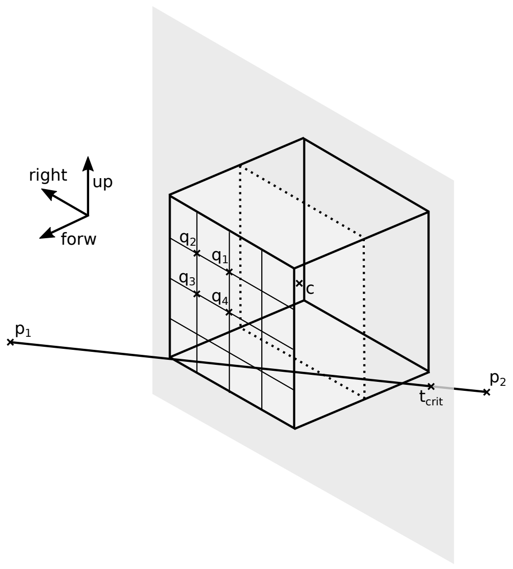


Figure 3.9: Explanation figure for alg. 4.

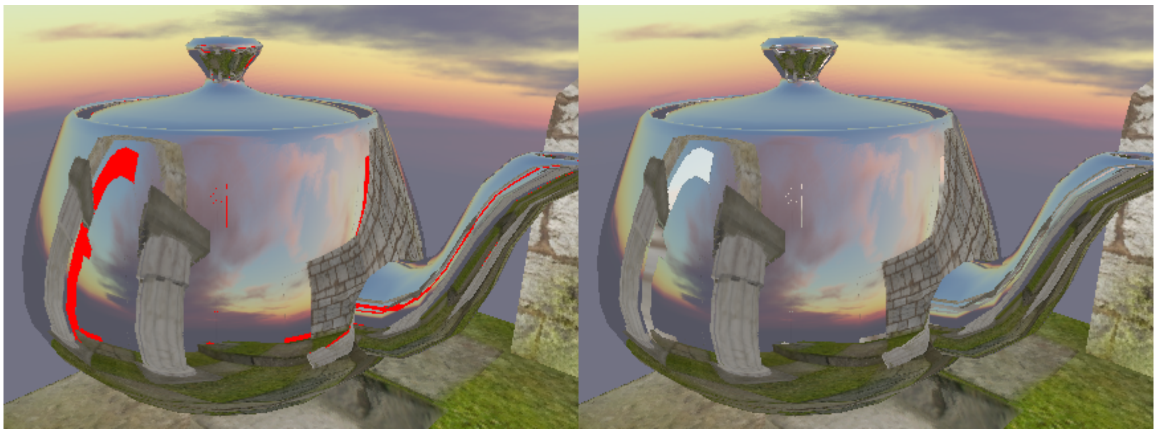


Figure 3.10: Filling the unresolved areas with environment mapping is fairly noticeable due to its low accuracy, which contrasts with the rest of the reflections.

Data:

$\vec{c}$	cubemap center point
$\vec{p_1}, \vec{p_2}$	start and end point of the traced ray
$\vec{q_1}, \vec{q_2}, \vec{q_3}, \vec{q_4}$	corner points of current acceleration cell
$\vec{up}, \vec{right}, \vec{forw}$	direction vectors of the current cubemap side

```
Algorithm correct_t($t$, $correct\_range$)
    if $t > correct\_range[1]$ then
        return $-\infty$
    else if $t < correct\_range[0]$ then
        return $\infty$
    return $t$

$t_{crit} \leftarrow line\_plane\_intersection(\vec{p_1}, \vec{p_2}, \vec{c}, \vec{c} + \vec{right}, \vec{c} + \vec{up})$

if $t_{crit} = \infty$ then
    $t_{range} \leftarrow (-\infty, \infty)$
else if $(\vec{p_2} - \vec{p_1}) \cdot \vec{forw} > 0$ then
    $t_{range} \leftarrow (t_{crit}, \infty)$
else
    $t_{range} \leftarrow (-\infty, t_{crit})$
end

$t_1 \leftarrow correct\_t(line\_plane\_intersection(\vec{p_1}, \vec{p_2}, \vec{c}, \vec{q_1}, \vec{q_2}), t_{range})$
$t_2 \leftarrow correct\_t(line\_plane\_intersection(\vec{p_1}, \vec{p_2}, \vec{c}, \vec{q_2}, \vec{q_3}), t_{range})$
$t_3 \leftarrow correct\_t(line\_plane\_intersection(\vec{p_1}, \vec{p_2}, \vec{c}, \vec{q_3}, \vec{q_4}), t_{range})$
$t_4 \leftarrow correct\_t(line\_plane\_intersection(\vec{p_1}, \vec{p_2}, \vec{c}, \vec{q_4}, \vec{q_1}), t_{range})$

$t_{next} \leftarrow max(min(t_1, t_2), min(t_3, t_4))$
$t_{prev} \leftarrow min(max(t_1, t_2), max(t_3, t_4))$

return $t_{next}, t_{prev}$
```

Algorithm 4: Algorithm for computing acceleration structure boundaries shown in figure 3.5. The computation of values $\vec{q_1}, \vec{q_2}, \vec{q_3}, \vec{q_4}, \vec{up}, \vec{right}, \vec{forw}$ is not included, for simplicity. The algorithm has to deal with eliminating intersections that are on the opposite side to the currently tested side of the cube. For this, an interval of allowed intersection values (in terms of t) is constructed (t_{range}). Function *correct_t* adjusts the computed intersection according to this interval. Figure 3.9 helps to visualize this algorithm. Alternatively, the approach described in section 3.7 could be used instead of this algorithm.

Chapter 4

Implementation

Along with this work come implementations of some above mentioned algorithms. It is provided as open-source software at GitHub[1]. The implementations include:

- `gl_wrapper` – simple C++ OpenGL-based rendering engine with example programs,
- examples of planar mirror and environment mapping written with `gl_wrapper` and
- the cubemap tracing algorithm (fragment shader version) written with `gl_wrapper`, with and without acceleration.

The compute shader version of the algorithm is only partially implemented and could not be tested. Also self-reflections aren't working as intended, mainly because the bias value is very difficult to set correctly and will probably require to be computed for each ray individually. The development was done on an ordinary laptop with NVidia GeForce GT 540M graphic card, under Ubuntu 16.04 operating system, with OpenGL version 4.5.0. Two CMOs with 256×256 textures were mostly used during development, with window size set to 640×480. Step length was empirically set to 0.001 (i.e. maximum of 10 000 samples per ray).

It is worth mentioning Blender[2], an open-source 3D modelling and rendering tool, which was used to model test scenes, render ray traced reference images, visualize the debugging information gathered from shaders and also to make the final video of the project. When using it, attention has to be paid to the fact that Blender uses a different coordinate system than OpenGL.

4.1 Engine

OpenGL in itself offers only low-level GPU API. In order to be able to work with high-level graphics concepts, such as scene management, usually an additional helper code has to be written. For the purposes of this work a simple engine, called `gl_wrapper`, has been created. Alternatively existing engines, such as OpenSceneGraph[3] or GPUEngine[4], could be used.

[1] `https://github.com/drummyfish/mirrors`
[2] `http://www.blender.org`
[3] `http://www.openscenegraph.org/`
[4] `https://github.com/Rendering-FIT/GPUEngine`

The engine is for the sake of simplicity of use written as a single C++ header file (similarly to for example glm library) and provides an object-oriented API with convenience methods and high level functionality methods. The key features include:

- singleton `GLSession` class that handles OpenGL initialization, basic input callbacks etc.,
- `Shader` class which handles shader loading and compilation, with additional support for `#include` directive in shader code,
- transformation classes for easy camera and vertex transformation matrix generation,
- `UniformVariable` class for easy work with uniform shader variables,
- `TransformFeedbackBuffer` class which allows retrieving vertex buffer output values, mostly for debugging purposes,
- `Image2D` class representing an image that can be used as a texture and saved/loaded to/from a PPM file,
- texture classes (`Texture2D`, `TextureCubeMap`) for easy texture management,
- `FrameBuffer` class for rendering to texture,
- `Geometry3D` class for easy 3D geometry management, with possibility to save/load to/from an OBJ file,
- convenience functions for creating basic geometrical shapes,
- `ReflectionTraceCubeMap` – a helper class for cubemap tracing algorithm,
- `CameraHandler` class with pre-programmed camera control,
- classes (`Printable`, `ErrorWriter`) for debugging and OpenGL error checking,
- `Profiler` class for performance measuring,
- `ShaderLog` class that serves as a support for debugging shaders,
- helper functions (for easy loading text from files, ...) and types (texel, ...),
- no additional library dependencies, aside from OpenGL, glew, freeglut and glm.

The library structure in form of class diagram is shown in fig. 4.4.

4.2 Basic Algorithm

The basic form of the fragment shader version of cubemap tracing is implemented in `cubemap` folder of the project repository. A custom scene (846 tris) made with Blender, a low-polygon version of Sponza (29705 tris, created from the original version also with the help of Blender) and an edited version of a scene[5] (4674 tris) found at Blendswap website were used for testing.

Algorithm 1 is implemented as a GLSL shader in `shader_quad.fs` file. It has not yet been heavily optimized.

[5]`http://www.blendswap.com/blends/view/63351`, provided under CC-BY by irokrhus

4.3 Acceleration

The computation of acceleration structure has been implemented both as CPU and GPU code with `compute_acceleration_texture_sw()` and `compute_acceleration_texture()` methods of `ReflectionTraceCubeMap` class, respectively. Example result is shown in fig. 4.1. The GPU version uses Frame Buffer Objects and a fragment shader. For each MIP map level $i > 0$ and for each cubemap side s a fullscreen quad is rendered, textured with s side image of MIP map level $i - 1$, invoking a fragment shader which samples four texels corresponding to each fragment coordinates, computes the distance minimum and maximum and writes them to the output color (R and G channels, B channel is used for mirror mask if needed). Alternatively, compute shaders could be used, but would require further GPU support.

Acceleration significantly reduces the number of iterations needed to find the intersection, as seen in fig. 5.1. However it improves the overall performance only by a small amount, probably due to the complexity of alg. 4 and/or inefficient implementation.

Computation of compute shader version of the acceleration structure has also been implemented with `compute_cs_acceleration_texture()` method, using compute shaders, as they are required for the algorithm itself. In current state it only works with 1024×1024 textures. The result is also stored in MIP maps as shown in fig. 4.2, but because of the different subdivision not all levels and texels are used. It is computed as follows.

For each level a dispatch of resolution corresponding to given level is invoked. Each workgroup of the invocation, consisting of a single thread, computes the minimum/maximum pair by iterating over the corresponding cell of 32 texels and stores the pair in the corresponding pixel. A more efficient way would be to have workgroups of 32 threads that would utilize `atomicMin`/`atomicMax` functions on shared variables, but these do not support floating point data. `GL_NV_shader_atomic_float` extension adds the support, but only for `atomicAdd` and `atomicExchange`[6]. The best way to optimize the process would be to use a parallel reduction with shared memory and thread synchronization.

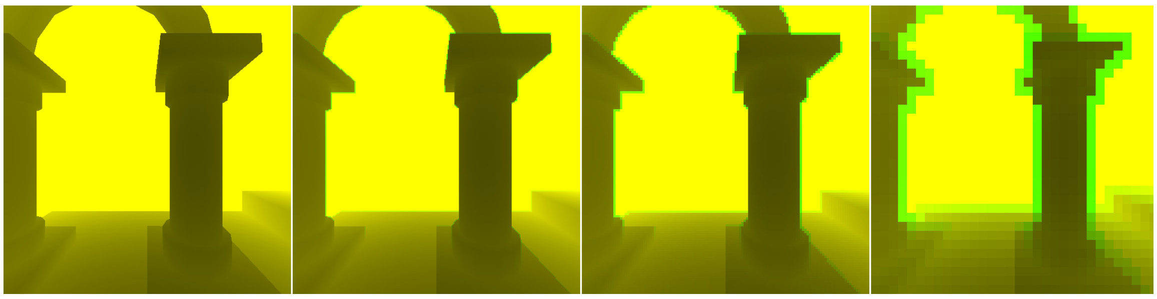


Figure 4.1: The acceleration structure is stored in MIP maps. Minima are stored in red channel, maxima in green. Here are shown MIP map levels 0 to 3 of one of the cubemap sides.

[6]https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_atomic_float.txt

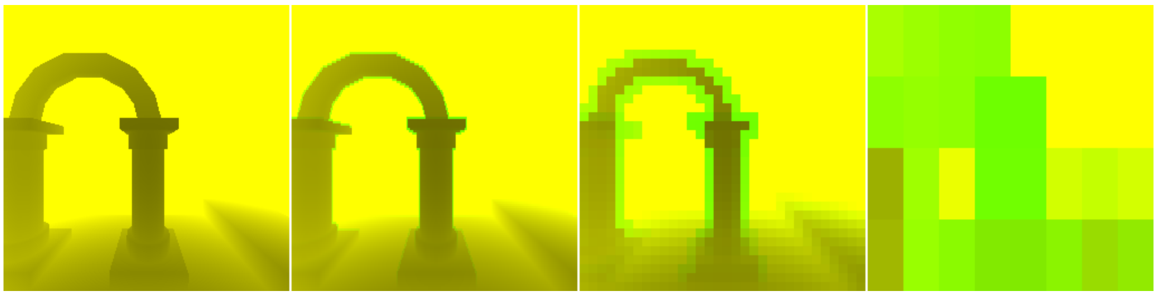


Figure 4.2: All four levels of one side of the acceleration structure for compute shaders, stored in the same way as the structure in fig 4.1, but subdivided as shown in fig. 3.6, from left to right: 1024 × 1024, 256 × 128, 32 × 32, 8 × 4.

4.4 Compute Shader Version

The compute shader version has not been completed. It was planned to use an OpenGL extension `NV_shader_thread_group`[7], which provides a convenient way of communication between the workgroup threads, in form of `ballotThreadNV` function. It allows each invocation to set one bit of a shared 32-bit value. This is well usable to distribute the cell mask. GPUs without support for this extension can use other means of communication, such as shared memory.

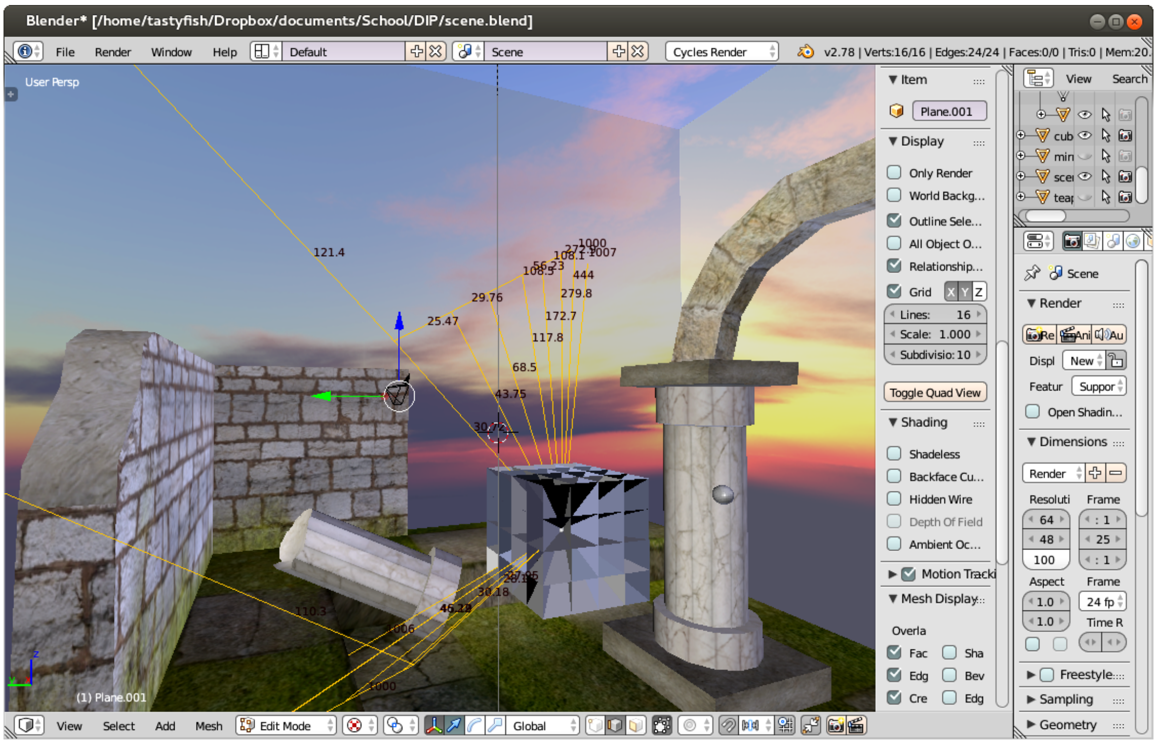


Figure 4.3: Blender was used to help visualize debugging information from shaders, convert and make test scenes, render reference images and other important tasks.

[7] `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_shader_thread_group.txt`

4.5 Debugging

Debugging complex shaders may be a challenging task because the OpenGL pipeline has been designed to transfer data from CPU to GPU, not the other way around. There is therefore no trivial way to send debugging info back from shader code. For this reason a debugging support has been programmed as a part of `gl_wrapper`. `ShaderLog` class uses Shader Storage Object Buffers (SSBOs) to allow GLSL shaders to write data to a debug log in a way similar to classic programs writing to standard output.

In order to use the shader log, the shader has to include (using `gl_wrapper`) a code in `shader_log_include.txt` file. This code will allow the shader to use logging functions and constants, such as `shader_log_write_vec3(...)`, `shader_log_write_char(...)` etc. As a next step an instance of `ShaderLog` class is created in the C++ program, which will allow the data to be retrieved from GPU and written out to standard output.

Shader code can also sometimes be tested on CPU with the glm library, which mimics the GLSL language. With shader logs available, Blender is a useful tool for visualizing the data in 3D space (fig. 4.3).

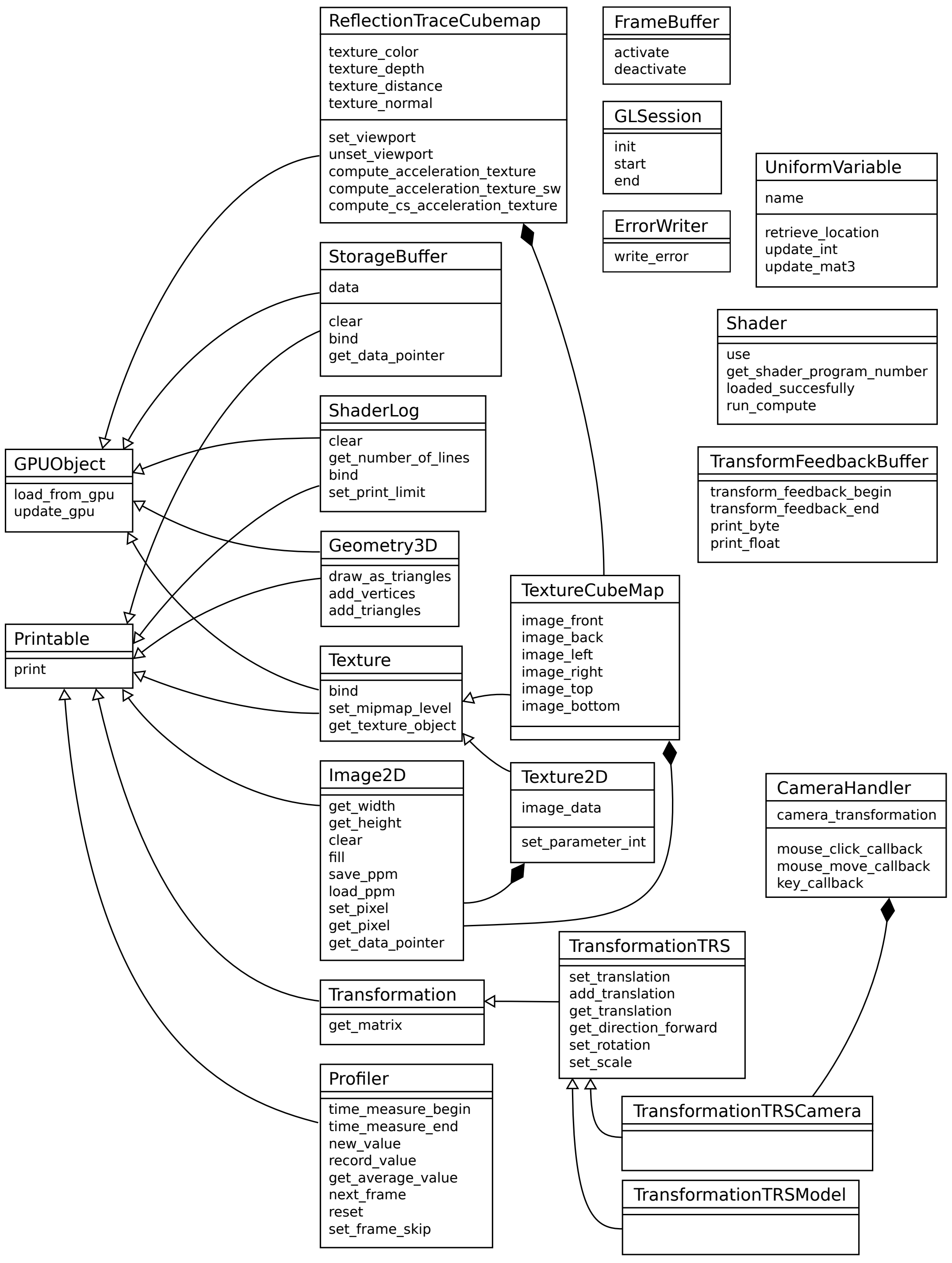


Figure 4.4: gl_wrapper class diagram

Chapter 5

Results

This chapter presents the achieved results of the work. Fig. 5.3 shows comparison of effects of different parameters. The results are comparable to ray tracing if CMOs are correctly placed (as described in section 3.8) and, with current implementation, if there are no self-reflections, which do not yet work correctly. The advantages of the resulting algorithm are mainly following:

- Its execution time is not correlated with scene or reflector geometry complexity.
- It is independent of the scene representation (e.g. triangle meshes, volumetric etc.); as long as we're able to render the cubemaps, the algorithm will work, as we're working with rendered depth profile, not the scene itself.
- The model of the scene (i.e. CMOs) is quite simple to recompute and does not take a great amount of memory, unlike light fields. This allows for reflecting partially dynamic scenes.
- It can be implemented with fragment shaders only. Use of compute shader will, however, probably greatly improve it.

5.1 Performance

This section presents the tested performance of the implementation on different platforms and with different parameters.

OpenGL query objects were used to measure performance. The `Profiler` class in `gl_wrapper` allows to measure time using `GL_TIME_ELAPSED` queries and number of rasterized pixels using `GL_SAMPLES_PASSED` queries. This actually measures the number of fragments that pass the depth test which may not be equal to rasterized pixels, but can be close enough with reasonably shaped geometry and backface culling turned on. (The exact number of rasterized mirror pixels could be counted with SSBOs.)

Table 5.1 shows a comparison of some of the existing methods described in section 2.3 and the new method. Table 5.2 shows the measured performance.

method	N	D	T	S	storage	vertex	pixel	geom. req.	refl.	env.
env. mapping	yes	no	no	no	low	low	low	no	none	low
virt. objects	yes	yes	yes	no	low	high	low	yes	high	mod.
light field map	yes	yes	no	yes	high	low	mod.	no	high	high
light field cube	yes	yes	no	no	high	low	mod.	no	low	high
cubemap tracing	yes	yes	yes*	yes*	mod.	low	high	no	low	mod.

Table 5.1: Updated comparison of existing mirror rendering methods as presented in the work of Jingyi Yu et al. [22], to include the new algorithm. Compared is support for near/distant/touching/self-reflections (N/D/T/S), texture storage demands, per-vertex/per-pixel computational demands, requirement for scene geometry and the cost of dynamic reflectors and dynamic environment. The table shows that cubemap tracing trades higher per-pixel computational demands for lower memory and precompute costs. * – The feature should be supported, but hasn't been fully implemented.

parameters	NVidia GT 540M	NVidia Titan X	AMD Radeon RX 480
320×240, 256, A, S0	18.4 FPS, 1.80 μs	485.2 FPS, 0.06 μs	152.6 FPS, 0.18 μs
640×480, 256, A, S0	7.4 FPS, 1.28 μs	198.6 FPS, 0.04 μs	69 FPS, 0.11 μs
320×240, 256, A, S1	29.2 FPS, 1.23 μs	612.8 FPS, 0.04 μs	321.2 FPS, 0.09 μs
640×480, 256, A, S1	9.6 FPS, 0.91 μs	284.8 FPS, 0.02 μs	120.8 FPS, 0.06 μs
320×240, 256, S0	21.4 FPS, 1.77 μs	500.4 FPS, 0.06 μs	226.8 FPS, 0.12 μs
640×480, 256, S0	7.8 FPS, 1.19 μs	229.4 FPS, 0.04 μs	111 FPS, 0.06 μs
320×240, 256, E, S0	4.2 FPS, 8.91 μs	165 FPS, 0.22 μs	62.8 FPS, 0.5 μs
640×480, 256, E, S0	1.6 FPS, 6.71 μs	78 FPS, 0.12 μs	26.6 FPS, 0.3 μs
640×480, 128, A, S0	5.4 FPS, 1.77 μs	141.8 FPS, 0.05 μs	55 FPS, 0.14 μs
640×480, 512, E, S0	8.8 FPS, 1.11 μs	180.2 FPS, 0.04 μs	74.8 FPS, 0.09 μs
640×480, 128, A, L, E, S0	11 FPS, 0.81 μs	456.6 FPS , 0.02 μs	139 FPS, 0.05 μs
640×480, 512, A, L, S0	10.4 FPS, 0.89 μs	232.8 FPS, 0.03 μs	77.8 FPS, 0.07 μs
320×240, 256, A, L, S0	31.6 FPS, 1.92 μs	654.6 FPS, 0.04 μs	202.4 FPS, 0.13 μs
640×480, 256, A, L, E, S0	13.2 FPS, 0.66 μs	505.8 FPS, 0.01 μs	189.6 FPS, 0.03 μs
640×480, 256, L, S0	21.2 FPS, 0.41 μs	536.2 FPS, 0.01 μs	225 FPS, 0.03 μs
640×480, 256, L, E, S0	2.6 FPS, 3.69 μs	140.2 FPS, 0.06 μs	45.4 FPS, 0.17 μs
640×480, 256, A, L, E, S, S0	5.2 FPS, 1.61 μs	202.8 FPS, 0.04 μs	75.4 FPS, 0.09 μs
640×480, 256, L, E, S, S0	2.2 FPS, 5.01 μs	100 FPS, 0.09 μs	30.2 FPS, 0.25 μs
800×600, 256, L, S0	14 FPS, 0.38 μs	N/A	155 FPS, 0.03 μs
128	8 ms, 7 ms	15 ms, 16 ms	6 ms, 1 ms
256	35 ms, 10 ms	49 ms, 19 ms	11 ms, 2 ms
512	136 ms, 14 ms	193 ms, 8 ms	31 ms, 2 ms

Table 5.2: Measured performance, parameters are in format: window resolution, cubemap resolution, flags (A – acceleration, E – efficient (optimal) sampling, L – analytical intersection, S – self-reflections, S0/S1 – scene). Measured metrics were FPS and average time per reflector pixel. The last three rows show times of CMO re-rendering and acceleration structure recomputation in milliseconds. The scene and camera settings were left at default values (the same as in fig. 5.3).

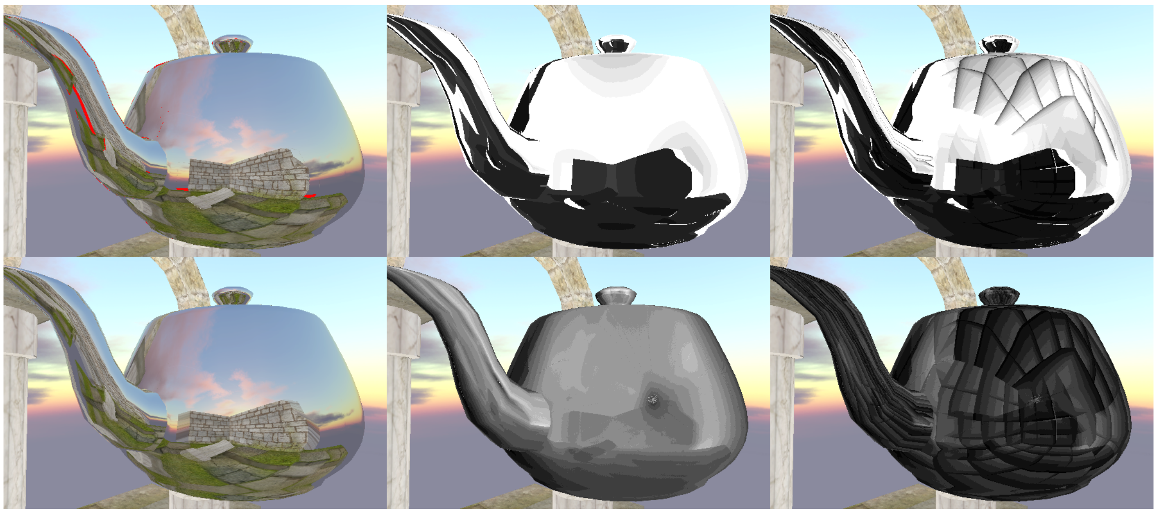


Figure 5.1: The figure shows the visual result (left) and the number of iterations needed to trace the ray for each pixel, without (middle) and with (right) acceleration. The top line shows distance threshold intersection criterion. The bottom line shows analytical intersection which allows for much shorter step length and much fewer iterations.

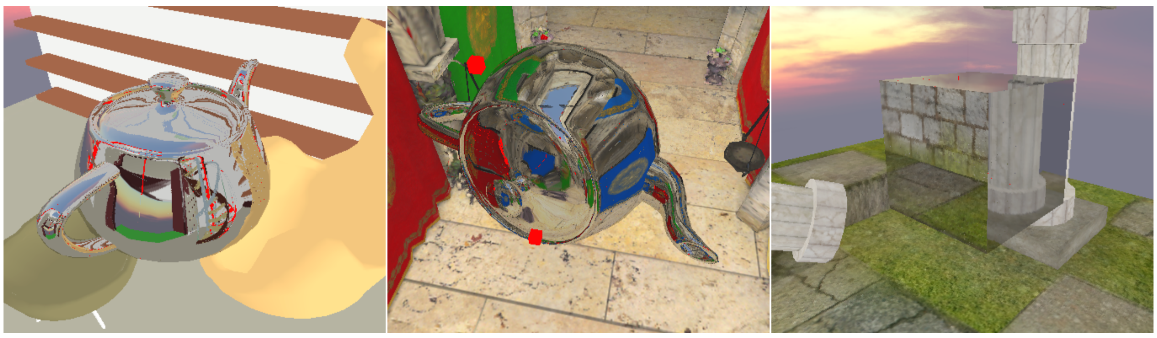


Figure 5.2: various results at the three test scenes

The biggest performance drop is usually caused by unresolved intersections that make the algorithm trace the whole ray from within all CMOs. Acceleration seem to help with this by allowing big skips over empty areas. Also using analytical intersection shows much better performance than distance threshold because it leaves almost no pixels unresolved and tolerates a bigger step size. Table 5.2 shows that the algorithm works at interactive rates on high-end GPUs, and with certain settings even on an ordinary GPU.

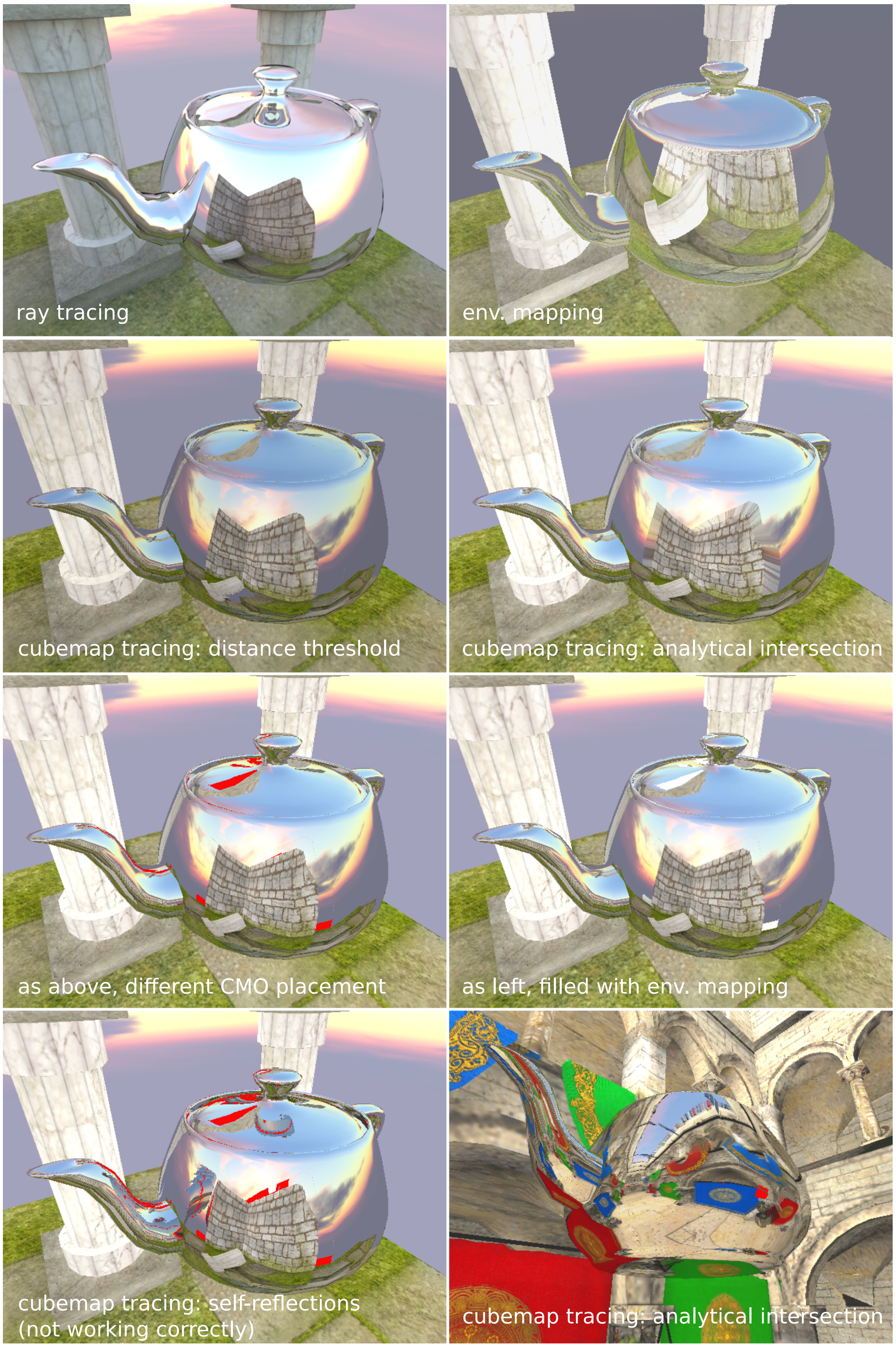

ray tracing
env. mapping
cubemap tracing: distance threshold
cubemap tracing: analytical intersection
as above, different CMO placement
as left, filled with env. mapping
cubemap tracing: self-reflections (not working correctly)
cubemap tracing: analytical intersection

Figure 5.3: results – comparison of different parameters

Chapter 6

Future Work

This work made only a few steps in one possible direction towards solving the interactive reflection problem and leaves many things for future research, including:

- completing the compute shader version of the algorithm,
- experimenting with greater number CMOs,
- using different scene-to-texture projections (eg. paraboloids),
- using different technologies (eg. CUDA, OpenCL, Vulkan, Direct3D etc.),
- using different acceleration structures,
- testing different ordered samplings and dynamic sampling strategy switching (as explained in section 3.5),
- addressing potential aliasing issues when reflections cause subsampling of the scene,
- automatically placing the CMOs in the scene,
- creating more efficient version of alg. 4 – the system of inequalities described in section 3.7 could probably be used,
- testing the best combinations of possible parameters (step length, sampling strategies, intersection criteria, . . .),
- constructing hybrid algorithms, i.e. combining with other methods.

Use of paraboloids instead of cubemaps could be worth testing. A single paraboloid captures only half the view of a cubemap [10], so either a greater number of them should be used or dual-paraboloid technique could take place. Paraboloids might more easily avoid conditional jumps in code, but the projection isn't rectilinear, so the rays would generally not be projected to straight lines. This would also affect the acceleration structure.

Another unaddressed issue are view-dependent effects, such as specular reflections. Only diffuse materials were used in this work so the issue wasn't apparent, but can happen as illustrated in fig. 6.1. The solution would be to apply deferred shading at the level of CMOs, i.e. the algorithm would, instead of directly returning color, return parameters of the intersected surface point (i.e. diffuse color, normal, position etc.) and leave the computation of the final color for later stages.

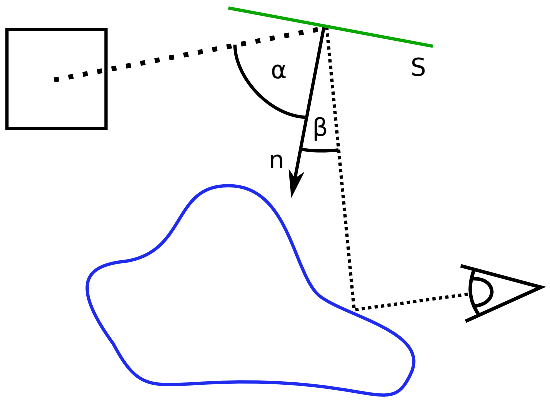


Figure 6.1: The viewer sees the reflection of the surface S. From their point of view view-dependent effects should be computed with respect to angle β, but the presented algorithms return the color computed with respect to the CMO, i.e. angle α, which is different. This can be solved by returning surface parameters instead of color.

Chapter 7

Conclusion

This work provided a summary of research on the topic of accurate interactive reflection rendering on non-planar mirrors and proposed a new algorithm, called cubemap tracing, based on capturing the scene from multiple positions into cubemap textures. It was implemented with OpenGL, accelerated, tested on different platforms and the results were evaluated. Possibly more efficient version of the algorithm, that would use compute shaders, was also described and is left for future implementation.

The advantages as well as disadvantages of the method lie in possibilities of parameterization – different parameters, such as step length or different sampling strategies, can be combined, the CMOs can be placed at different places in the scene etc. In agreement with initial expectations, the algorithm still has to make tradeoffs and so the ultimate method for reflection rendering remains yet to be found, and will probably also have to wait for the evolution of graphic hardware.

The algorithm implementation remains in a state suitable for research and testing but is in many ways unfinished and not much usable in practice. Despite the efforts put into debugging, the code still contains bugs and inefficiencies and isn't very robust. The reason is a lot of time was spent on developing the framework for the application and trying out many blind alleys while designing a brand new algorithm, but also that the main goal of the work was to experiment rather than to create a robust implementation. The presented algorithm nevertheless supports many desired features and shows great potential.

The work could continue by making a better, more stable implementation, integrating it into a widely used engine and testing its usability in practice. The current implementation is already able to run at interactive rates, with certain settings even on low-end GPUs, and so we can suppose that the more elegant and potentially more efficient compute shader version would greatly improve on it and offer very fast execution.

The contribution of this work is mainly in exploration of one of many possible research paths and documenting the encountered basic issues that have to be dealt with. A new algorithm was presented that can be placed next to the existing ones so that developers have more options to choose from when faced with the problem of reflection rendering.

Bibliography

[1] Bentley, J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM*. 1975.

[2] Blinn, J. F.; Newell, M. E.: Texture and reflection in computer generated images. *Communications of the ACM*. 1976.

[3] Cabral, B.; Olano, M.; Nemec, P.: Reflection Space Image Based Rendering. *SIGGRAPH '99*. 1999.

[4] Crassin, C.; Neyret, F.; Sainz, M.; et al.: Interactive Indirect Illumination Using Voxel Cone Tracing. *I3D '11 Symposium on Interactive 3D Graphics and Games*. 2011.

[5] Deering, M.; Winner, S.; Schediwy, B.; et al.: The triangle processor and normal vector shader: a VLSI system for high performance graphics. *SIGGRAPH '88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. 1988.

[6] Eisemann, E.; Décoret, X.: Fast Scene Voxelization and Applications. *I3D '06*. 2006.

[7] Greene, N.: Applications of World Projections. *Journal IEEE Computer Graphics and Applications*. 1986.

[8] Greene, N.; Kass, M.; Miller, G.: Hierarchical Z-buffer visibility. *SIGGRAPH '93 Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993.

[9] Heidrich, W.; Lensch, H.; Cohen, M. F.; et al.: Light Field Techniques for Reflections and Refractions. *EGWR'99 Proceedings of the 10th Eurographics conference on Rendering*. 1999.

[10] Heidrich, W.; Seidel, H.-P.: View-independent Environment Maps. *HWWS '98 Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. 1998.

[11] Kilgard, M. J.: Improving Shadows and Reflections via the Stencil Buffer. 1999. [Online; accessed 21-May-2017].

[12] il Kweon, G.; Hwang-bo, S.; hee Kim, G.; et al.: Wide-angle catadioptric lens with a rectilinear projection scheme. *APPLIED OPTICS*. 2006.

[13] Lee, W.-J.; Shin, Y.; Lee, J.; et al.: Real-Time Ray Tracing on Future Mobile Computing Platform. *SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications.* 2013.

[14] Lengyel, E.: Oblique View Frustum Depth Projection and Clipping. 2005.

[15] Ofek, E.; Rappoport, A.: Interactive Reflections on Curved Objects. *SIGGRAPH '98.* 1998.

[16] Popescu, V.; Sacks, E.; Mei, C.: Sample-Based Cameras for Feed Forward Reflection Rendering. *IEEE Transactions on Visualization and Computer Graphics.* 2006.

[17] Saito, T.; Takahashi, T.: Comprehensible rendering of 3-D shapes. *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques.* 1990.

[18] Segal, M.; Akeley, K.: *The OpenGL Graphics System: A Specification (version 4.5).* 2016.

[19] Sintorn, E.; Olsson, O.; Assarsson, U.: Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes. *SIGGRAPH Asia 2011.* 2011.

[20] Tze-Yiu; Wan, L.; Leung, C.-S.; et al.: Unicube for Dynamic Environment Mapping. *IEEE Transactions on Visualization and Computer Graphics.* 2011.

[21] Wald, I.; et al.: State of the Art in Ray Tracing Animated Scenes. 2009.

[22] Yu, J.; Yang, J.; McMillan, L.: Real-time reflection mapping with parallax. *Proceedings of the 2005 symposium on Interactive 3D graphics and games.* 2005.