FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Rasterized Curved Reflections in Screen Space

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Attila Szabo
Matrikelnummer 0925269

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.-Ass. Dipl.-Ing. Reinhold Preiner
Mitwirkung:

Wien, 06.04.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Rasterized Curved Reflections in Screen Space

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Attila Szabo
Registration Number 0925269

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Ass. Dipl.-Ing. Reinhold Preiner
Assistance:

Vienna, 06.04.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Attila Szabo
1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Das Berechnen von Reflexionen auf spiegelnden Objekten ist ein wichtiger Teilbereich in dem Gebiet der Bildsynthese in der Computergrafik. Reflexionen helfen dabei, bestimmte Materialeigenschaften im finalen Bild darzustellen und unterstützen den Betrachter dabei, Objekte und die Distanzrelationen zwischen ihnen zu erkennen. Mit der stets wachsenden Wichtigkeit von Computersystemen im Alltag entsteht auch ein Interesse an Methoden, die solche Reflexionen für die Nutzung in interaktiven Systemen wie Computerspielen und Virtual Reality bei interaktiven Bildwiederholraten berechnen können.

In dieser Arbeit wird eine gegebene State-of-the-Art Methode und zwei für sie mögliche Erweiterungen untersucht. Die Methode ist dafür konzipiert, eine akkurate Reflexion von Geometrie auf der Oberfläche eines gekrümmten Reflektors in Echtzeit zu berechnen, und dabei die Fähigkeiten der Rendering Pipeline, die auf moderner Grafikhardware implementiert ist, auszunutzen. Ihr Funktionsprinzip baut auf der Idee auf, für jeden Eckpunkt einer Geometrie den Reflexionspunkt auf der Oberfläche des Reflektors zu finden und die Grafikhardware die reflektierte Geometrie mit diesen neuen Punkten rasterisieren zu lassen.

Zwei bedeutende Probleme, die dabei auftreten, bestehen darin, dass die Suche nach den Reflexionspunkten einerseits sehr zeitaufwändig sein kann, und andererseits die lineare Interpolation, die bei der Rasterisierung durchgeführt wird, Artefakte auf der gekrümmten Oberfläche des Reflektors zur Folge hat. Die erste Erweiterung, die in Betracht gezogen wird, hat das Ziel, die benötigte Zeit zum Finden eines Reflexionspunktes zu reduzieren, indem eine hierarchische Datenstruktur und ein entsprechend abgeänderter Suchalgorithmus verwendet werden, wobei die Daten aber gleich effizient gespeichert werden wie zuvor. Die zweite Erweiterung versucht den Bildfehler der linearen Interpolation zu verringern, indem die reflektierte Geometrie tesselliert wird. Die Tessellation erfolgt adaptiv auf Grund einer Fehlermetrik die auf dem reduzierten Bildfehler basiert. Abschließend werden einige Resultate, gefolgt von einigen möglichem Denkanstöße für zukünftige Forschung in diesem Feld, vorgestellt und diskutiert.

Abstract

The rendering of reflections on mirror-like objects is an important operation performed in image synthesis. Being able to calculate the reflections on reflective surfaces in a rendered scene helps visualize many materials which have such properties and aids the viewer in recognizing objects and the perception of distance relations between them. Considering the increasing use of computer systems in day-to-day life, there is much interest in implementing methods that are able to render these reflections at interactive framerates for use in interactive systems, such as computer games and virtual reality.

In this paper one given state-of-the-art method and two possible extensions are examined. The method is designed for rendering accurate reflections of geometry on the surface of a curved reflector, utilizing the capabilities of the rendering pipelines implemented on contemporary graphics hardware, in real-time. It is based around finding the reflection point for each vertex of a geometry object, and then letting the graphics hardware rasterize the reflected geometry using the found points.

Two important problems with this approach are that the search for the reflection point can take a long time, and that the linear interpolation used in the rasterizing step leads to artifacts on the reflector's curved surface. The first examined modification is aimed at reducing the time needed for the search of a reflection point by using a hierachial data structure, and an accordingly different searching technique, while still storing the data as efficiently as before. The second modification attempts to reduce the linear interpolation error in the final image by tessellating the reflected geometry. This is done adaptively based on a metric for the error reduced. Finally, some results are presented and discussed and some ideas for possible future work in this field is given.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The rendering of visually pleasing imagery from virtual scenes has been a major working field of the discipline of *Computer Graphics* (CG) since its beginning several decades ago. It is oftentimes the goal to portray materials and physical effects in a life-like fashion. Such physical effects are observed in the real world and many methods and tools have been developed to include them in a CG system. These methods and tools do not necessarily aim for maximizing *realism*, the physical accuracy of the depicted phenomenon, but often may instead go for *believability*, meaning that the result will often only be ”correct enough” for it to look relatively accurate to the viewer while possibly gaining faster performance, being easier for a designer to

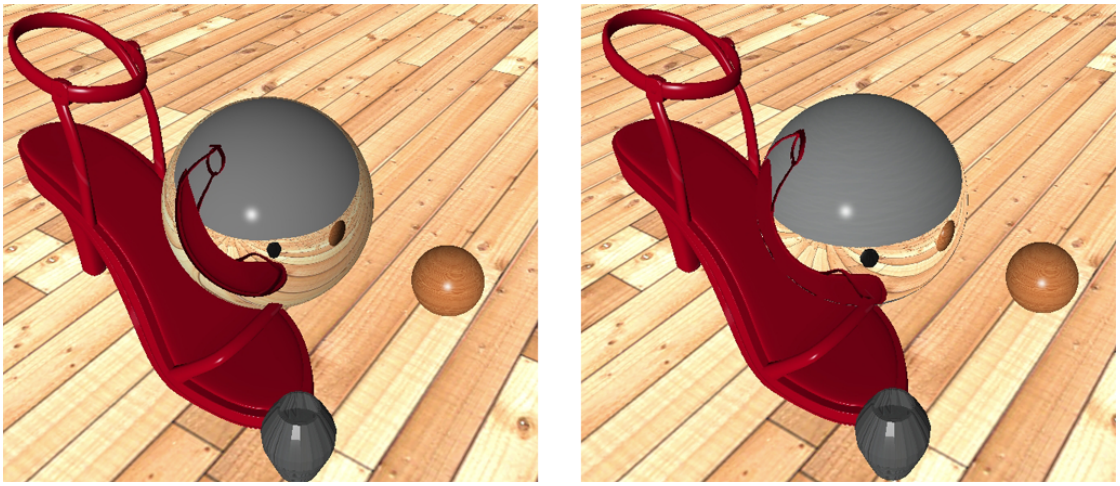

Figure 1.1: *Left:* Approximate reflection using environment mapping with noticable errors. *Right:* Accurate reflection.

Figure 1.2: An example for a complex scene rendered with ray tracing. Image taken from [16].

work with, looking aesthetically more pleasing, etc. Popular and well-documented examples for approximate rendering methods include *Phong Shading* [9] and *Bump Mapping* [1]. On the other hand, *Ray Tracing* [15] is a method for rendering various physical effects very accurately.

There are several physical effects of general interest to be accounted for in a rendering system. Assigning materials to objects, such as household materials, natural materials, metals, glass and materials with micro-structures on the surface, makes the scene feel believable to the viewer. The rendering of atmospheric effects, for example fire, smoke, clouds, fog or atmospheric scattering, and the simulation of a lighting model for the scene are also important for immersing the viewer in a scene. The goal is to portray effects such as light bouncing off surfaces into the scene, light penetrating into translucent objects and scattering under the surface, or light being reflected off of an object or refracted when passing through an object. In the following, one solution for the rendering of mirror-like reflections on a surface is presented and discussed.

1.2 Rendering Reflections

Rendering of highly reflective surfaces, such as mirrors, has been possible already relatively early in the history of computer graphics, around the 1980s, when *Ray Tracing* [15] had been first described. In this approach, the rendering process simulates a number of viewing rays being emitted into the scene from the position of the virtual camera, each ray originating from a pixel of a virtual screen, and then for each viewing ray calculates the result of the interactions with the surface the ray hits. To simulate a mirror-like reflection, this approach may generate additional reflection rays originating from the point where a viewing ray intersects with the surface of an object and send them out into the scene. The object hit by a reflection ray will finally be seen as a reflection on the surface of the mirror. The result of this approach is a very realistic image of a virtual scene, an example can be seen in Figure 1.2, and many extensions to this approach have been developed to be able to simulate a high number of other physical phenomena.

The main disadvantage of *Ray Tracing* is its comparatively high computational cost. There may be a very high number of viewing rays and each ray has to be intersected with the scene

Figure 1.3: An example for reflection mapping being used in the movie *Flight of the Navigator*. Image taken from [13].

many times. The search for these intersections may take a very long time. If the rendering happens *offline*, the required production time may not pose a significant problem. However, *online* rendering, the purpose of which is to continuously simulate a virtual world and provide interactive framerates, usually implies heavy performance constraints in order to retain interactivity. Even though consumer graphics hardware has seen a significant increase in possible performance over the recent years and improvements to make the algorithm work effectively on graphics hardware have been proposed [10], *Ray Tracing* is still not always suitable to provide interactive framerates in many cases.

Several alternative methods exist with the goal of improving performance to interactive framerates. One such method is *Reflection Mapping*, in which an approximation of the reflection can be found very rapidly. It has been used as part of CG effects in commercial film for the first time around the 1980s [13]. In this approach, an environment is stored within a texture, such as a *cube map* where a data structure resembling the six sides of a cube is used. The environment can pre-rendered, taken from a series of photographs, drawn by hand, etc. The environment is stored in the six faces of the cube map as if the view point was in the center of the cube. Afterwards, when rendering, for each viewing ray, the reflection direction on the surface of the mirror is calculated and then transformed into a texture coordinate for lookup in the cube map. This way, the expensive simulation of reflection rays in the scene is replaced by a computationally very cheap texture lookup. The resulting image is a believable depiction of a reflective object which can be computed very quickly. This method lends itself well for application in interactive simulations and has been successfully used in movies and computer games. An example can be seen in Figure 1.3. The major drawback of this approach is that the reflection is not physically accurate, as it works under the assumption that all of the light reaching the surface of the reflective object comes from an infinite distance. No parallax occurs in the reflected scene and therefore the reflected objects are always slightly off. This effect is especially noticable with objects close to the mirror or intersecting it, an example can be seen in Figure 1.1.

Finally, a popular approach to create reflections on a planar mirror encompasses drawing the scene twice - once normally and once from a virtual viewpoint derived from the mirroring plane. The mirror image is then placed over the mirror in the original image. To this end, the scene needs to be rendered from the mirror's "virtual" viewpoint first. To find this virtual viewpoint,

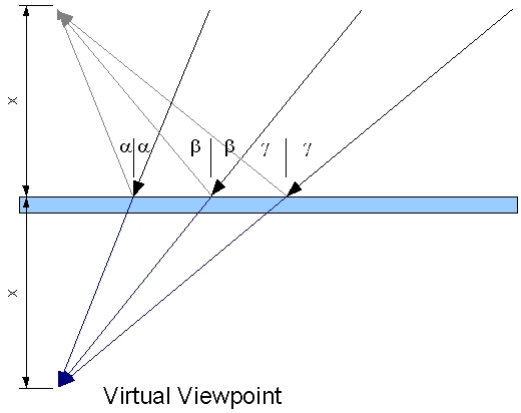

Figure 1.4: The virtual viewpoint from which a planar mirror's virtual scene can be rendered, as constructed by finding the reflection vectors' convergence point. Image taken from [7].

the reflection rays for some points are calculated on the surface of the mirror plane using the law of specular reflection. The virtual viewpoint then lies at the position where these reflection rays converge [7]. Figure 1.4 illustrates this property. Once the virtual viewpoint has been found, the scene is rendered from that viewpoint and the resulting image is inverted. This inverted image is then applied as surface texture for the planar mirror in the second rendering pass, in which the scene is drawn from the original viewpoint. In the final image, the mirror shows the correctly reflected scene. This approach has been used many times in commercial interactive software, particularly computer games, with great success [5]. The big drawback of this technique is its limitation to planar mirrors: the virtual viewpoint's position is not the same for every mirror surface point anymore if the mirror is curved.

1.3 Contents of this Thesis

The approaches discussed in this paper aim to address all three of the previously mentioned methods' drawbacks. Firstly, the reflection should be able to be calculated for curved mirrors, in particular spheroid and ellipsoid mirrors which may be convex or concave. Secondly, the reflection should be rendered accurately, based on the detail of the scene's geometry and the approaches' base assumptions, where possible. Thirdly, the reflections should be able to be calculated in real-time, resulting in a technique suitable for use in interactive environments.

This work examines a proposed technique for rendering reflections on mirror-like reflective objects with curvature, as proposed by Estalella et al. [3]. This approach is based on the idea of rasterizing the reflected scene in an additional rendering pass and finding the correct geometry transformations using a screen-space search. These operations are performed efficiently on the GPU. Furthermore, possible extensions to this technique aimed at improving the performance and enhancing the resulting image quality will be discussed.

In the first part of this paper (3), a system following the concepts from [3] is implemented and described. The system first renders a mirror's surface points' positions and normals in screen

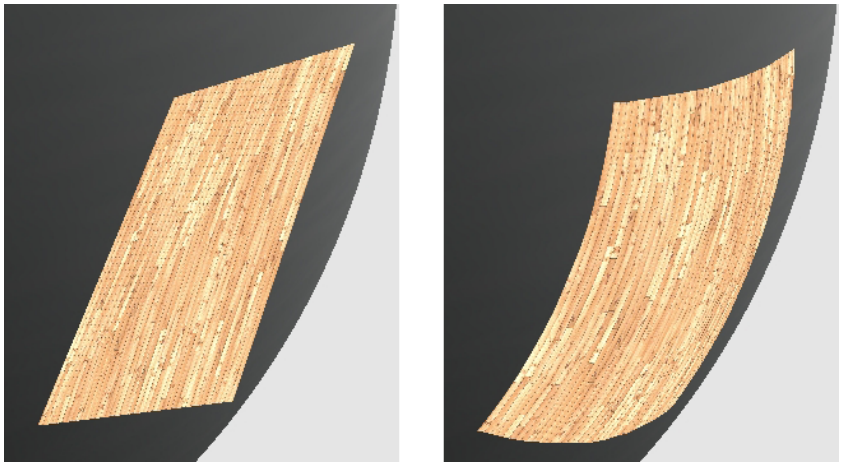

Figure 1.5: *Left:* Erroneous reflection of a square. Its sides are linear interpolations between the corners' reflections. *Right:* What the reflection should look like.

space. Then, for each vertex in the scene, it uses a linear search to find the pixel that comes closest to its actual reflection on the mirror. For the result, the scene is rendered once normally without the mirror and once with the vertices transformed to their corresponding reflections. The two images are then blended to yield the final image containing the scene with the mirror.

In the second part (4), an alternative search method is examined. This approach groups a mirror's surface points together to form a hierarchy in a bottom-up fashion. The reflection of a vertex is found by traversing the hierarchy top-down. After the reflections have been found for all vertices, the final image is rendered as before.

In the third part (5), an extension is proposed for improving the visual quality of the rendered image. The rasterizer linearly interpolates between the reflected vertices to render the reflection. This inherently leads visual errors in the final image since the reflection of a straight line (triangle edge) on a curved reflector should yield a curved line. The problem is exemplified in Figure 1.5. To adress this problem, the virtual geometry of the reflected scene is tessellated and the newly created vertices are placed correctly on the reflector surface, increasing the curvature of a reflected edge. Furthermore, since the increased number of reflection point searches becomes computationally expensive relatively quickly, an attempt to lower the performance impact is made by tessellating adaptively. An error value is calculated for all geometry primitives and the tessellation is only performed for those whose error value exceeds a threshold. The result is a rendered reflection on the curved reflector in which low-polygon geometry shows less of the linear interpolation error.

In the final part (6) follows a discussion of the systems' results and a comparison of the relative differences in performance and visual quality. In addition, properties and problems of the systems are highlighted and possible solutions as well as future work for added functionality and improved performance are suggested. The paper ends with a concluding summary.

CHAPTER 2

Related Work and Algorithms Overview

2.1 Multi-Pass Pipeline Rendering

The multi-pass approach for rendering a mirrored scene from some ”virtual” viewpoint and its subsequent composition with the rendered image of the original scene has been examined before, for example in [2]. The approach stems from the idea to reduce the large amount of computation required by *Ray Tracing* methods to deliver an accurate reflection by utilizing commercial graphics hardware's optimized performance capabilities for vertex-based geometry. Furthermore, by using operations and operating primarily on data structures which are already used for the original rendering pass, the method remains free of large computational overhead from possibly having to transform data or perform precomputations before the rendering step, as may be the case in alternative rendering systems with comparable results [2]. The approach is inspired by the physical-world observation of the image in planar mirrors corresponding to the original scene as seen from a different viewpoint, an effect that may be used by interior designers through covering walls with mirrors in order to make a room appear larger. The reflected image may be merged with the image of the reflector by applying it as a texture or rendering it directly to the screen using a stencil buffer [8].

2.2 Curved Reflectors

Extensions for reflectors which show curvature have been proposed, such as in [8]. In the case of curved reflectors, the rendered image of the reflected object appears deformed compared its original appearance, since no single ”virtual” viewpoint for use in a second rendering pass can be found across the curved reflector surface. Instead, every point to be reflected has to be transformed differently. This approach realizes this concept directly by transforming the scene vertices individually to their virtual counterparts and then proceeding to use their connectivity

information to triangulate the virtual objects. The search for the virtual points' locations is the major focus of this approach. The goal of this search is to find the appropriate primitive from the reflector's surface, which is represented by polygonal geometry (such as triangles), and to use that primitive to perform the reflection for the point to be mirrored. The reflection is performed by mirroring the vertex across the reflector's tangent plane, which is interpolated from the triangle corners' tangent planes. This particular technique implements an efficient data structure to accelerate the search, the so-called *explosion map*, which basically resembles a mapping of all the reflector's primitives' IDs to a surrounding environment map-like spherical data structure. The ID of the corresponding reflector primitive for a point to be reflected can be found using the structure, and the primitive with that ID is then used for the reflection. While this method is efficient, it adds computational overhead because in dynamic scenes the *explosion map* has to be re-calculated every time the reflector or the viewpoint are moved. However, it well illustrates how the calculation of a reflected image on the surface of a curved reflector can be made much more performant by experimenting with and choosing more appropriate data structures.

2.3 Linear screen-space Search

In this paper, an implementation of the system proposed by Estalella et al. [3] is discussed. The system utilizes different data structures and the rasterizer to make the reflection search and the rendering more efficient. The approach works by utilizing the information about reflector surface points' positions and their normals to perform a searching algorithm to find the correct reflection point. For every reflector the reflection is rendered and then blended with the original image. First, the scene's mirror is rendered into a geometry buffer, storing the surface positions and normals. Then, the reflected location for each of the scene's vertices is found and the reflected scene is rasterized. To find the reflected position for a vertex, the following geometric principles are utilized [3]:

Consider a vertex V of world geometry that is to be reflected and the virtual camera's viewpoint O. For every point P on the surface of a curved reflector ρ the *bisector vector* B_P of the angle formed by the observer location O, the surface point P and the world vertex V can be defined, as well as the curved reflector's *surface normal* N_P. The *point of reflection* R on the surface of the curved reflector is such that its bisector vector B_R and its surface normal N_R coincide. To identify this case, the condition whether $N_R \cdot B_R$ equals 1 can be evaluated for the reflector surface point R. Figure 2.1 visualizes this principle. This point of reflection is unique across closed convex reflectors [4].

The search for one reflection point is performed by a linear search in screen space using the previously acquired geometry buffer. First, a starting point is chosen, a good choice is the center of the reflector object's projection (or, simpler, its projected bounding box). Then, the four neighbouring texels are examined in a crosshair-pattern and their bisector and normal vectors retrieved and their dot products computed. The next step of the search then shifts the crosshair's center to the neighbour which has the largest dot product and the new neighbours are examined. This search continues until the dot product of one (in practice, the largest dot product) has been found. The pixel location of the reflection point in the current frame can be used as a starting

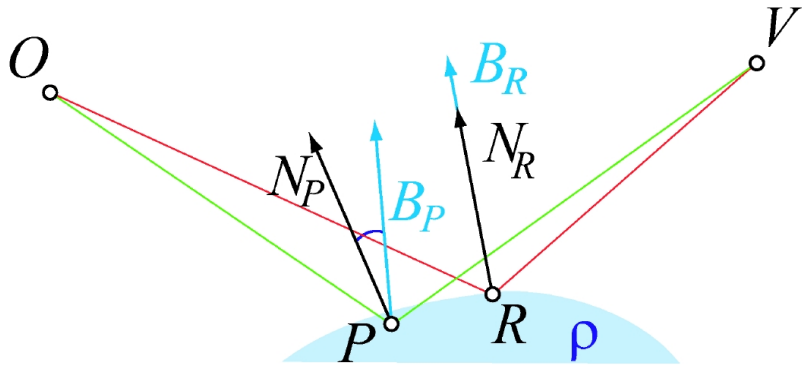

Figure 2.1: The relation between the viewpoint O, the world vertex V and a point on the reflector surface. The reflection point is such that their bisector vector and the surface normal coincide. Image taken from [3].

Figure 2.2: On the left: An example scene rendered with the system of Estalella et al. [3]; on the right: the same scene rendered with ray tracing. Image taken from [3].

point on the next frame's search for the reflection point in order to take advantage of the frame-to-frame coherence typically present in interactive systems.

The search is performed at the fragment shader stage of the rendering pipeline and then the resulting points are stored in a buffer object in order to enhance its speed by utilizing the graphics hardware's capability for parallelism. When the search has been performed for all the original scene's vertices, their topology information is used to connect the virtual vertices to virtual objects which compose the virtual scene. After the original scene has been rendered, the virtual scene is drawn on top of it to render the reflection.

The resulting system renders scenes with reflective curved convex objects at interactive framerates on typical consumer hardware of 2006. Furthermore, it has the big advantage of being capable of handling scenes with dynamically moving objects without any loss of performance.

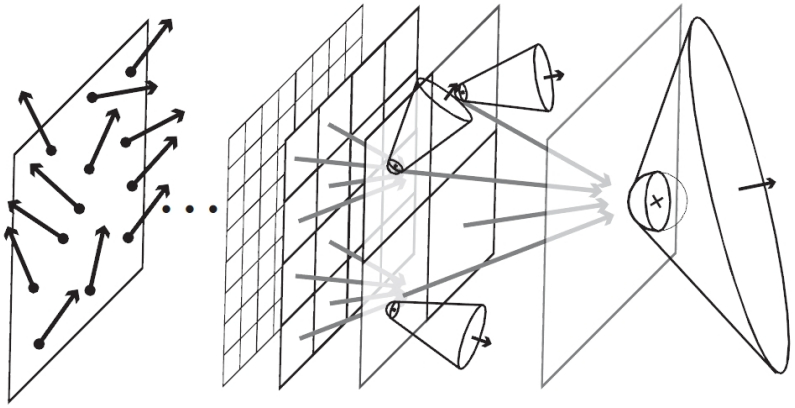

Figure 2.3: The ray hierarchy bundling rays together in sets of spheres and cones as performed in the system by Roger et al. [11]. Image taken from [11].

On the other hand, performance decreases with each reflector in the scene, since the entire calculation has to be repeated for each reflector. The quality of the reflection also depends greatly on the detail (number of vertices) of both the scene geometry and the reflector surface, as well as some minor artifacts occurring at the edges of a reflector. An example scene rendered with this system can be seen in Figure 2.2.

2.4 Hierarchical Search

An improvement to the search methodology is insipired by the ray tracing extension proposed by Roger et al. [11]. This approach uses a hierarchical data structure for an accelerated simulation of viewing rays' interaction with the scene that can also be applied to the searching algorithm used in the previous approach. The data structure bundles viewing rays together such that their points of origin are grouped together an enclosing sphere and their corresponding ray direction vectors are grouped an enclosing cone. Then, these spheres and cones are hierarchically grouped together into a new set of enclosing spheres and cones. This process is repeated until the hierarchy of rays reaches its uppermost level, in which one sphere and cone remain which encompass all rays. A visualization of this process can be seen in Figure 2.3. This particular system builds such a hierarchy to group together viewing rays whose interactions with the scene need to be simulated in a ray tracing technique. When traversing this hierarchy, scene geometry can be quickly excluded from consideration in the upper hierarchy levels, effectively eliminating the need for simulation for a large subset of viewing rays, which in turn speeds up the entire rendering process considerably.

In this paper, such a ray hierarchy data structure is adapted for use in the searching algorithm discussed in the previous approach. The goal is to create an image-space hierarchy which can be traversed to find the reflection of a vertex in fewer steps than with the linear search. Again using only one geometry buffer storing a mirror's surface information, the hierarchy is constructed in

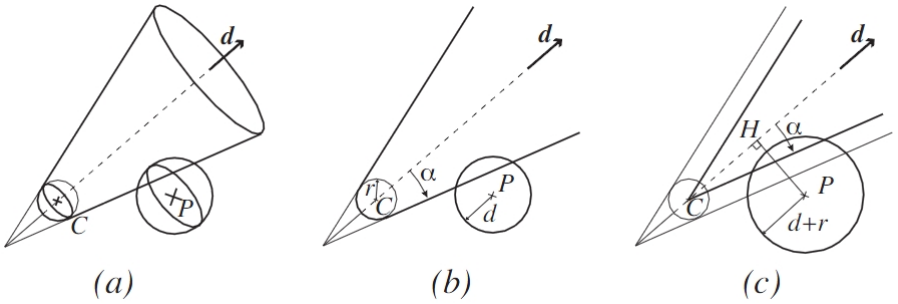

Figure 2.4: *(a)* The cone sphere intersection test *(b)* reduced to a 2D problem and *(c)* performed with the data available in the ray hierarchy. Figure taken from [11].

a way resembling mipmap generation. The radii of the spheres and opening angles of the cones are equal to zero at the bottom-most level of the hierarchy. In a bottom-up approach, the next level of the hierarchy is generated by grouping together the four pixels ”below” one pixel. The new values store the average of the four sphere origins and cone directions, and a radius and cone opening angle which enclose the lower four cones and spheres.

Concretely, the geometry buffer stores a mirror’s surface positions and reflection rays, which are the directions of the viewing rays reflected off the mirror’s surface. To find the reflection of a vertex, the hierarchy is traversed starting at the single cone in the top level. In each level, the cone containing the vertex is chosen, until the bottom level is reached. The result of this search is the reflection ray that ”hits” the original world point exactly (or, in practice, comes closest to it) and its point of origin is the reflection.

To check whether or not an object interacts with a set of rays, an intersection test between the object’s bounding sphere and the cone has to be performed. Because of the symmetry of revolution, this operation is a 2D intersection test [11]. Furthermore, to perform the test with the information given by the hierarchy, the apex location of a cone with direction d and opening angle α can be assumed to be it’s sphere’s center point C when the world object P bounding sphere’s radius d is enlarged by the radius of the sphere corresponding to the cone r. A diagram of this concept can be seen in Figure 2.4. At this point, a simple cone sphere intersection test using this data yields the sought result.

2.5 Adaptive Tessellation

Finally, a related work by Roger et al. [12], comparable to the previous approach of multi-pass rendering, uses a searching method by attempting to converge towards the reflection point faster through utilizing variable step sizes from examining the gradients of light travel paths. Importantly, this work also raises questions about the quality of the curved reflection since the graphics hardware interpolates linearly between reflected vertices on a curved surface. This property produces very noticable visual errors when a large object’s geometry uses very few vertices, for example a large square or cube defined only by their corner vertices. The reflected

edges are straight lines although they should be curved on a curved mirror. An example for the error can be seen in Figure 1.5.

This paper therefore examines the possibility of tessellating reflections dynamically and especially using a form of adaptive tessellation. The goal is to tessellate only those parts of the geometry which show especially noticable visual errors. To that end, the previous approaches are extended by the following technique. An object's reflection is tessellated to increase the number of vertices along its edges. Since this may not always improve the image, an error metric is introduced to quantify the benefit of tessellation. Tessellation is only performed for those parts of the reflection where the error is significantly reduced.

CHAPTER 3

Accurate Reflections using a Linear Screen Space Search

In this chapter, a system for rendering the reflection on a curved reflector based on the principles presented by Estalella et al. [3] is implemented. The system is a multi-pass rendering approach and uses a linear screen-space searching algorithm to find the reflection point for a world geometry vertex. This implementation uses the *OpenGL 3.3* rendering API, but no specific functions will be mentioned.

3.1 Implementation

The system assumes the scene to consist of geometry objects, which consist of *triangle* primitives, and which are marked to be *mirrors* or *non-mirrors*. Each scene object's vertices have a world-space position and the normal vector of the surface they belong to. The functionality of the system is outlined by the following pseudo-code given in Algorithm 3.1.

```
1 foreach NonMirror n do
2 |   Draw(n);
3 end
4 foreach Mirror m do
5 |   gBuffer_m ← RenderPositionsNormals(m) ;
6 |   foreach NonMirror n do
7 |   |   DrawReflection(gBuffer_m,n);
8 |   end
9 end
```

Algorithm 3.1: Functional outline of how a frame is rendered.

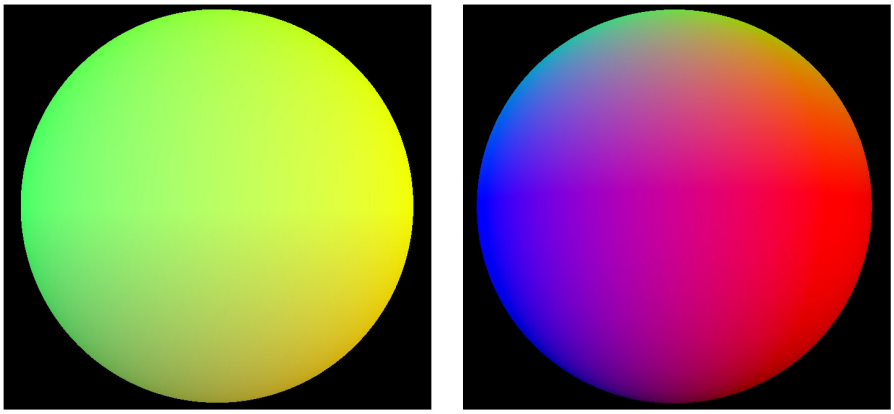

Figure 3.1: An example for a sphere's surface positions (*left*) and normals (*right*), both interpreted as RGB color components.

Given the two groups of geometry objects of mirrors and non-mirrors, all the non-mirrors are drawn normally first. An optional addition to this first step is to draw the mirrors as well, albeit with some darker or more background-colored shading, to make them visible without any reflection and to give the impression of some "ambient reflections".

Afterwards, in the pseudo-code function *RenderPositionsNormals*, for each reflector, the world-space coordinates of each surface point of the reflector and their corresponding surface normals are rendered into two 2D textures with four floating point components each. These textures are called the *Position Map* and the *Normal Map* respectively. The rendering is done from the same camera as the original scene. Render-to-Texture functionality of the rendering API is used to specify the two textures which are to be used as render targets. This process is straightforward since the reflector's vertices are simply transformed the same as the rest of the scene using the Model View Projection Matrix in the vertex shader stage of the rendering pipeline. Then the output values are calculated by transforming the vertices' positions and their corresponding surface normals into world space (by applying the Model Matrix to the positions and the transposed inverted Model Matrix to the normals [14]). Afterwards, in the fragment shader stage, for each pixel the interpolated output value is written to the corresponding texture. The result are the Surface Map and the Normal Map which contain the surface coordinates and normals of a reflector in world coordinates at pixel accuracy. An example for these textures is shown in Figure 3.1.

Following that, for each non-mirror scene object, its reflected geometry is found on the reflector surface and drawn by the pseudo-code function *DrawReflection*. This is the main part of this system and its functionality is displayed in the following Algorithm 3.2.

For one scene object, in principle, each of its geometry's vertices are transformed to a new position corresponding to the location of its reflection on the surface of the reflector. The object is then drawn normally using these new vertex positions. The search for the reflection point position is performed by the function *FindReflectionPoint*. The function starts its search at a

```
1  function DrawReflection(gBuffer, object)
2  foreach Vertex v of object do
3  |  v.Position ← FindReflectionPoint(gBuffer, v);
4  end
5  Draw(object);

6  function FindReflectionPoint(gBuffer, v)
7  CurrentPixel ← MirrorCenter();
8  repeat
9  |  PreviousPixel ← CurrentPixel;
10 |  CurrentPixel ← LeastErrorNeighbour(CurrentPixel,left,top,right,bottom);
11 until PreviousPixel == CurrentPixel;
12 return GetPosition(gBuffer , CurrentPixel);
```

Algorithm 3.2: The rendering step in which the geometry of a non-mirror's reflection is found on the surface of a mirror. The function *FindReflectionPoint* repeats a pixel-by-pixel step in the direction of least error until no longer possible.

certain texel of the reflector surface, and then iterates towards the reflection point texel. The center of the reflector in screen space is used as the starting point. For the currently considered texel, the algorithm examines its four directly neighbouring texels and calculates their reflection error. The method for calculating the reflection error for one texel is given in Algorithm 3.3, in which *S* and *N* are the position and the normal of the mirror surface point stored in this texel, *po* is the direction vector from the surface point to the camera and *pv* is the direction vector from the surface point to the original vertex.

```
1  function GetPixelError(gBuffer, Pixel)
2  S ← GetPosition(gBuffer,Pixel) ;
3  N ← GetNormal(gBuffer,Pixel) ;
4  po ← normalize( CameraWorldPosition - S);
5  pv ← normalize( VertexWorldPosition - S) ;
6  bisector ← normalize( po +pv) ;
7  return dot( bisector, N);
```

Algorithm 3.3: The method for calculating the reflection error of one pixel.

The error is given by the dot product between the bisector vector between these two direction vectors and the reflector surface normal. The searching algorithm calculates this reflection error for all four neighbours and the current texel and then simply considers the neighbouring texel with the lowest reflection error (highest dot product result) for the next step. This process is repeated until no neighbouring texels have a lower reflection error than the currently considered texel (all neighbours' dot product results are lower than the current texel's), at which point the

surface point in the current texel is returned as the final reflection point for the vertex.

This algorithm is performed in the vertex shader stage of the rendering pipeline. After the new positions for the vertices have been set, the modified geometry can be rasterized. To ensure correct visibility, the z-buffer needs to be updated with appropriate values. For each vertex, this value is the distance between the vertex and its reflection. This technique is well suited for utilization on contemporary graphics hardware since it has to make only few modifications and leaves much of the work to well-established and optimized rendering routines.

The reflected scene is drawn on top of the initially rendered image of non-mirrors. If several mirrors are involved, the reflected scenes are computed for each mirror in turn, the search algorithm also having to be performed for each mirror separately, which quickly lowers the performance of this approach with each additional mirror. Furthermore, this approach does not reflect a mirror off of another mirror, a case in which considerations of reflections inside of reflections would have to be made. The space where the reflection of another mirror should be can simply be left blank, or it may be replaced by a non-mirror version of the mirror geometry.

It is noteworthy that no special consideration is given to vertices which do not have their reflection point on the visible surface of the reflector. For a reflector with closed uniformly convex geometry, such as a sphere, these are the vertices hidden behind the projection of the reflector in screen space. Triangles containing such vertices are simply not drawn. The condition for identifying such a vertex is [3]:

$$pv \cdot N < 0$$

The dot product between the reflection direction and the surface normal is greater than zero for vertices with visible reflection points, equal to zero for vertices with the reflection point at the exact edge of the reflector and it becomes less than zero as the vertex moves behind the reflector. Vertices for which this condition is true are marked and their corresponding triangles are discarded in the geometry shader stage of the rendering pipeline. In addition, for reflectors which are not closed, such as reflectors that are partially obscured, the search terminates simply at the edge of the reflector projection at the highest dot product value [3]. This approach causes only minor visible errors if the geometry is tessellated finely enough, although additional considerations regarding this limitation are briefly outlined in [3].

3.2 Result

The result is a system that renders the reflection on the surface of a mirror by drawing the original geometry with transformed vertex locations. Those new locations are found through a linear pixel-by-pixel search in screen space across the projected image of the reflector using the law of specular reflection. The system is integrated seamlessly into the rendering pipeline implemented on modern graphics hardware. Furthermore, only two textures with four floating-point components of the same size as the regular rendered image are needed. All the computations are performed for each frame on a per-vertex basis, therefore the system does not suffer in performance in scenes with dynamically moving objects, making it suitable for use in interactive systems. Some examples for reflections computed by this system are given in Figure 3.2.

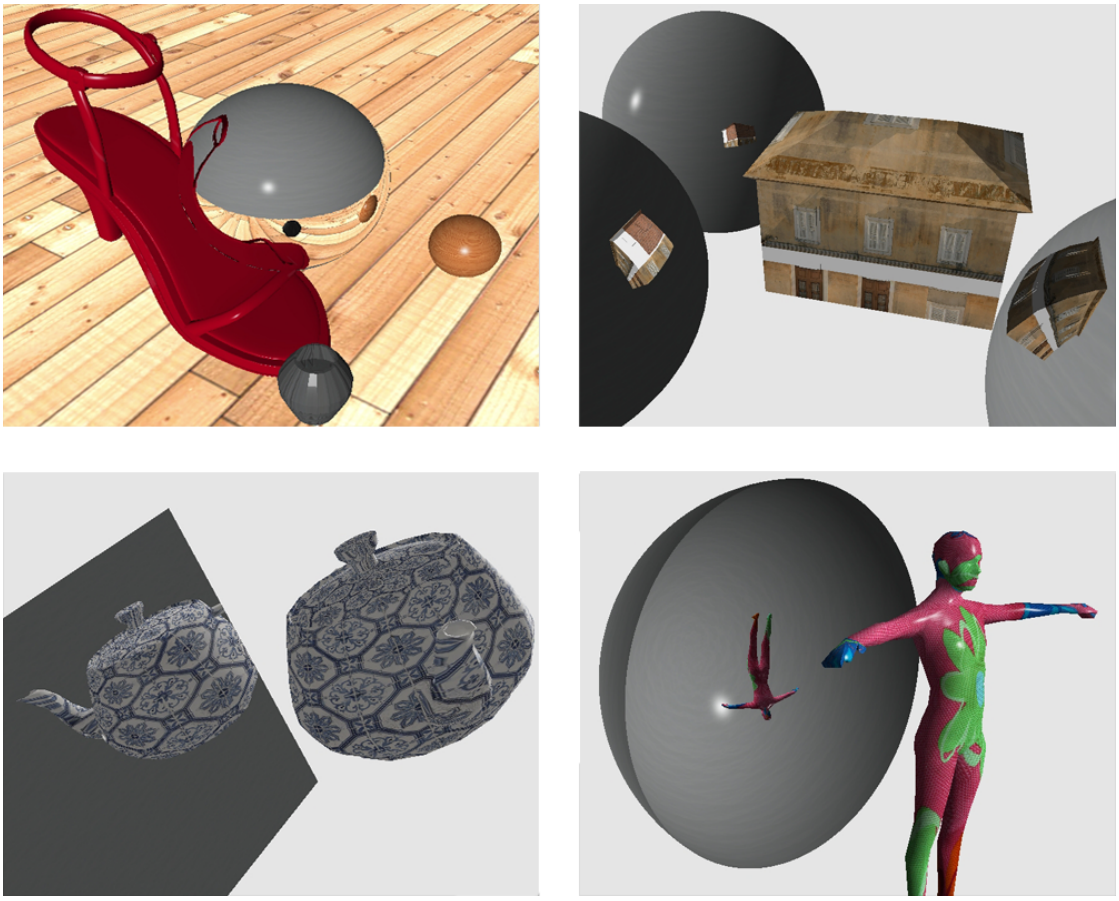

Figure 3.2: Examples for reflections rendered with our system. *Top left and top right*: Geometry reflected off one and off multiple mirror spheres. *Bottom left*: A reflection on a planar mirror. *Bottom right*: A reflection on a concave mirror.

The examples shown here include a reflection on a concave mirror, a case which requires special consideration due to its geometric nature. A concave mirror produces one reflection of an object rightside-up if the object is close enough, one reflection upside-down if the object is far enough away, and several possible reflections in the area in-between [12]. This approach handles vertices that fall into the first two categories fine, but cannot compute the multiple reflections correctly for a vertex from the third category. In addition, a concave mirror may produce self-reflections, which are not considered in this approach, or multiple reflections of one vertex at certain angles when the light would bounce around inside the mirror, in which case only the first reflection is calculated.

The other examples show some general settings in which geometry is reflected off spherical mirrors and a planar mirror. In the case of the planar mirror, the result is comparable in visual quality to the specialized approach using the reflected virtual viewpoint discussed in the Introduction (1.2).

CHAPTER 4

Reflection Ray Hierarchy Based Search

In this chapter, a modification to the system described in the previous chapter (3) is implemented and examined. This modification groups pixels together into a hierarchy. The goal is to require fewer steps for the search algorithm to terminate. With the hierarchical data structure comes the necessity for a different set of calculations when performing a step of the search.

4.1 Basic Idea

The basic idea is to group pixels together to form a hierarchy. By traversing this hierarchy, the search for a reflection point can converge in fewer steps compared to the pixel-by-pixel search in the previous chapter (3).

A conceptual sketch of the hierarchy is shown in Figure 4.1. The mirror's *reflection rays* [Fig. 4.1: *black arrows*], which are the viewing rays originating in the eye point reflected off its surface, are considered. To find the reflection of a point, the goal is to find the (closest) reflection ray that "hits" the point [Fig. 4.1: *red*]. Neighbouring reflection rays are grouped together using tight-fitting bounding cones [Fig. 4.1: *green*]. These cones are grouped together again using bounding cones, so long until only one cone remains [Fig. 4.1: *blue*]. This creates a hierarchy in which the final cone encompasses all reflection rays. The search for the reflection of a point starts at the largest cone. Subsequently, the next-smaller cone is selected if it contains the point, so long until the search arrives at a reflection ray, which yields the searched-for reflection.

The actual reflection is the reflection ray's point of origin on the mirror's surface. When constructing the hierarchy, the reflection rays' points of origin are also grouped together by the same principle, but using tight-fitting bounding spheres [Fig. 4.1: *orange circles*]. The center of a sphere corresponds to a cone's point of origin.

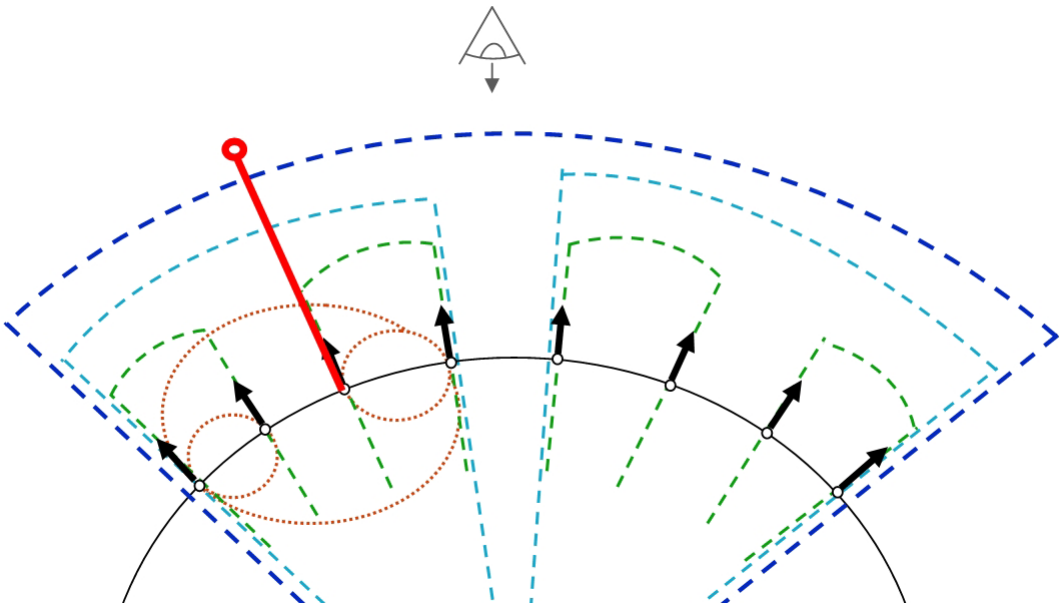

Figure 4.1: The concept of the reflection ray hierarchy. *black arrows:* reflection rays. *green and blue:* cones. *orange circles:* some of the spheres. *red:* a point and its reflection ray.

4.2 Implementation

The basic structure of the approach remains the same. First, all of the non-mirror geometry is rendered. Then, for each reflector two textures are rendered from the same camera as the original image, which form the bottom level of the ray hierarchy. The first texture stores the world coordinates of the surface points on the reflector in the x, y and z components of each texel. When the hierarchy is constructed, this texture represents the spheres and the w component stores the sphere's radius, or -1 if there is no sphere at that point.

The surface normal and the world position of the camera are used to calculate the reflection ray of the viewing ray emitted by the camera location in the direction of the surface point. Most rendering APIs provide an optimized `reflect` function for rapid calculation. The *OpenGL* implementation of the function is used for this system. The reflection vector is given by

$$S = 2(N \cdot V)N - V$$

where S is the reflection ray, N is the surface normal vector pointing away from the surface and V is the normalized incident vector from the camera world position to the reflector surface point. The resulting vector is written to the x, y and z components of the second texture. In the hierarchy, this texture holds the cones, with the w component storing a cone's opening angle or -1 if there is no cone there.

After this step, the hierarchy is built using these two textures as a base. Each texel of a level in the hierarchy corresponds to the four texels directly "below" it in a 2x2 window. For easier reading, these four texels are henceforth referred to as its *children*. To be able to find exactly four

children for each texel unambiguously, the pixel size of the base texture needs to be a power of two in both dimensions. To this end, the next largest power of two is calculated for the sizes of the current viewport in both dimensions, and the larger of the two results is then used as the size of the square render targets for the base of the hierarchy pyramid. Using square textures ensures that the hierarchy can be built without ambiguity problems and using a texture larger than the original viewport guarantees that no information is lost at the pixel-accurate result of the search.

The technique for building the hierarchy is given in the following Algorithm 4.1.

This algorithm is performed for both of the two base textures. Starting from the hierarchy level above the base, a new value is written for each texel in a bottom-up approach, calculated from its four children. Each texel represents a bounding cone or a bounding sphere in the hierarchy.

The first function call `GenerateMipmap` is a function provided in most rendering APIs for generating the texture objects used as the levels of the mipmap pyramid on the graphics hardware. In this case, the mipmaps are simply empty. Next, an iteration over each texel of each mipmap level above the base calculates the respective bounding geometry around its children.

To that end, the four children values in the level below are first located. Afterwards, the tight-fitting bounding sphere or bounding cone is calculated from these children. This is done for two children each in the function `boundSphere`, which calculates the sphere center `c` between the two centers and the radius `r` large enough to encompass both spheres. The function `boundCone` similarly calculates the cone axis direction `d` and opening angle `a` such that the two cones are enclosed. For the four children, the function `CalculateBounding` processes two children each, and then both their results.

The result of this algorithm is the ray hierarchy pyramid stored in the mipmap levels of its base texture, such that each texel contains the smallest sphere containing its child spheres in the position texture, or the smallest cone containing its child cones for in reflection ray texture.

It is noteworthy that in the way the hierarchy is constructed, there is no guarantee that the cones cover the entire reflection space originating from the reflector. Indeed, some cones' volumes may overlap, and some areas which have to be reflected may lie outside of any cone. In addition, at the edge of a convex reflector, at grazing refleciton angles, the spread between cones increases very quickly compared to the center of the reflector. The resulting bigger space between cones leaves a higher chance for such "holes" to appear. This poses special considerations necessary during the search algorithm and for its performance.

Continuing on to the next step, the vertices of the scene geometry now each have to receive their reflection position. Using the two textures that store the ray reflection pyramid, the reflection position for one world vertex is found by the following searching Algorithm 4.2.

This algorithm traverses the hierarchy pyramid in a top-down approach, starting at the only texel at the top of the pyramid and in each level, choosing the child texels for which their reflection cones contain the world vertex. At the bottom level of the hierarchy, the cones have an opening angle of 0, which is why no such test is possible and the reflection vector that has the smallest normal distance to the world vertex is simply chosen.

A stack structure is used in the algorithm for the cases in which the world vertex falls into multiple cones. In such a case, the algorithm "branches out" at the cones, effectively performing a depth-first search across each of the branches until a path to the bottom level is found.

```
1  GenerateMipmap() ;
2  for mipmap level l ← min + 1 to max do
3  |  foreach Texel on l do
4  |  |  Texel← CalculateBounding(Texel.children) ;
5  |  end
6  end

7  function CalculateBounding(children)
8  s1 , s2 , s3 , s4 ← children.spheres;
9  c1 , c2 , c3 , c4 ← children.cones;
10 sr ← boundSphere( boundSphere( s1, s2) , boundSphere( s3, s4)) ;
11 cr ← boundCone( boundCone( c1, c2) , boundCone( c3, c4)) ;
12 return sr , cr;

13 function boundSphere( sa , sb)
14 if sa.r = −1 then
15 |  return sb ;
16 end
17 if sb.r = −1 then
18 |  return sa ;
19 end
20 d ← sb.c − sa.c;
21 snew.c ← sa.c +0.5 · ( d + d/|d| · ( sb.r − sa.r )) ;
22 snew.r ← (|d |+sa.r +sb.r ) · 0.5 ;
23 return snew;

24 function boundCone( ca , cb)
25 if ca.a = −1 then
26 |  return cb ;
27 end
28 if cb.a = −1 then
29 |  return ca ;
30 end
31 cnew.d ← ( ca.d + cb.d )/| ca.d + cb.d | ;
32 cnew.a ← acos( ca.d · cb.d) ·0.5+ max(ca.a ,cb.a) ;
33 return cnew;
```

Algorithm 4.1: The method for building the ray hierarchy using the base texture and storing it into its mipmap pyramid.

```
1  curTexel ← startTexel;
2  stack.push(startTexel);
3  while ! stack.isEmpty() & curTexel.level > 0 do
4  |   curTexel ← stack.pop() ;
5  |   foreach child in curTexel.children do
6  |   |   if contains(child, worldVertex) then
7  |   |   |   stack.push(child);
8  |   |   end
9  |   end
10 end
11 curTexel ← closestDistance(curTexel.children, worldVertex);
12 return curTexel.sphere.center;
```

Algorithm 4.2: The method for finding the reflection point of a world vertex.

As discussed in Chapter 2, the test for a cone containing the word vertex is equal to a cone-sphere intersection test. In this test, the cone is the reflection cone of this child, and the sphere is given by the world vertex as its center and the current child's sphere's radius as its radius. If the cone and the sphere intersect, the child is pushed onto the stack. If they do not intersect, the child's reflection cone does not contain the world vertex and it is not considered any further.

After the hierarchy has been traversed to the bottom level, the resulting sphere center, which represents the reflection point on the surface of the reflector, is returned. Then the geometry can be drawn by the rendering pipeline as before. An example for the path taken by the search algorithm as the result of the step taken in each level can be seen in Figure 4.2. The image exemplifies how the sphere centers of the chosen children at each level of the hierarchy do not necessarily move closer to the actual reflection point, and the search path may have to move across large areas of the reflector.

Problems are posed if the world vertex lies in one of the special regions created by the cone hierarchy. The case of a world vertex being inside multiple cones at one level is handled well by the use of the stack structure, since *any* way through the hierarchy will result in the output being exactly at or close to the actual unique reflection point on the reflector. However, the case in which the world vertex lies in the "empty" space outside of any cone is more problematic. In this case, the algorithm in its current form will terminate before reaching the base level and return some sphere center from a level above as the result, which is usually far off from the actual reflection point. This special case may happen at any level of the hierarchy excluding the top and the base, but in experiments it increases in frequency drastically at the bottom half of the hierarchy pyramid. An example for a resulting sequence of images containing the effects of this circumstance is best described by having the reflection "mostly" correct on the screen, but singular vertices "jumping around" the screen as the camera is slowly moved.

Some basic modifications are applied to the search in order to lessen the errors caused in the final image by this circumstance. Firstly, the amount of vertices falling into holes between cones is higher in the lower levels of the hierarchy. Because of that, it is possible to let the search

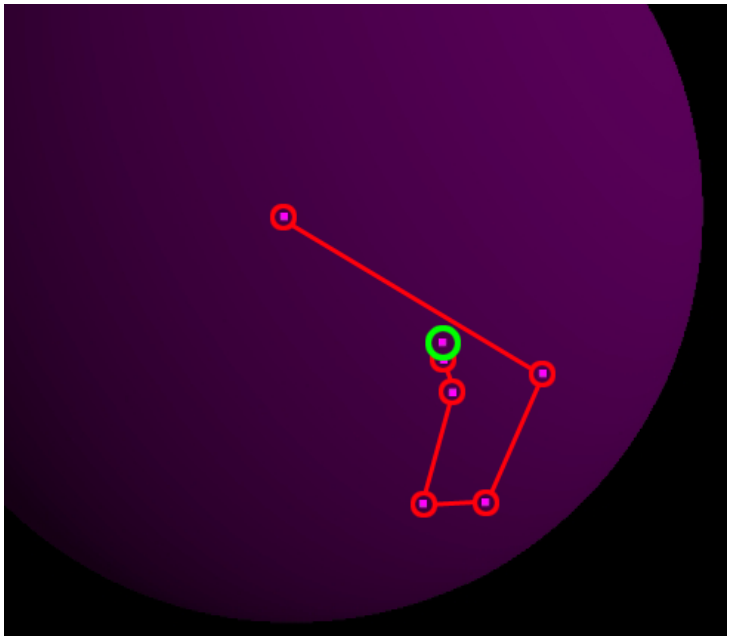

Figure 4.2: The path taken by the search algorithm to find the reflection point for one world vertex. The final reflection point is circled in green.

terminate before reaching the lowest level. The returned reflection points are then given by the center of the corresponding sphere that contains all the spheres in the levels below. As seen in Figure 4.2, those may not be even close to the actual reflection points for the upper levels of the hierarchy pyramid, but they are relatively close to it in the lower levels, introducing only small visual errors if this modification is chosen. However, experiments show that at least a few bottom levels have to be bypassed in a hierarchy pyramid in order to be free of the errors caused by the holes. This makes the reflection very erroneous. In addition, the error becomes larger at grazing angles, with a noticable jittering happening to the reflection between frames.

The other basic solution to the problem consists of expanding the volumes of the cones and the spheres used during search if the search does not terminate normally. This expansion has to happen up to the point where the world vertex lies inside at least one of the cones. The amount of expansion however is not easily gauged and the expansion requires additional computation. We solve this problem by introducing two flat factors, one for the opening angles of the cones and one for the radii of the spheres, by which the corresponding values are multiplied should the search fail. The condition whether or not the bottom level of the hierarchy has been reached is checked at the end of the search. If not, the last search step is repeated with enlarged cones and spheres. This approach eliminates practically all errors caused by the holes if the enlargement parameters are chosen sufficiently highly, but in turn introduces new inaccuracies in the final result. After the enlargement, it is no longer possible to say which of the cones leads closest to the actual exact reflection point anymore, and it may happen that the search arrives at a neighbouring point close to the accurate reflection point. The errors are most noticable at grazing

reflection angles, such as near the edge of a reflective sphere. In addition, with the factors becoming larger, the performance of the search may decrease as a result of the world vertex falling into more enlarged cones, some of which have to be checked but provide no path to the bottom. Experimentation with the factors has shown good values to be 1.5 for the cone enlargment factor and 2 for sphere enlargment. However, this is a simple approach and the error can still be reduced, for example by finding dynamic enlargment criteria.

4.3 Result

The resulting system has the ability to produce the same images as the previous system discussed in chapter 3. Previously used examples rendered with the system can be seen in Figure 4.3. However, it has proven to perform slower in the majority of cases [Figure 4.3 *(a)* and *(b)*]. First of all, the selection test that is performed in each iteration of the search algorithm is a test for intersection between a sphere and a cone. This operation is mathematically significantly more complex than the simple dot product calculated in the previous approach. In addition, this operation is performed at least once per level of the hierarchy pyramid, but usually more often. This number is not necessarily less than the number of search steps in the previous approach, especially if the starting point of the search is carried over to the next frame to utilize the frame-to-frame coherence. Such a frame-to-frame optimization is not possible in the present system. If one of the special cases of the relation between a world vertex and a set of cones from the hierarchy applies, performance generally becomes worse. If the world vertex is inside multiple cones, the search branches off and possibly examines the children of every one of those cones, increasing the number of intersection tests. If the world vertex lies in the empty zone between cones, the search step has to be repeated with enlarged cones and spheres, again increasing the chance of the vertex falling into multiple enlarged cones [Figure 4.3 *(c)*]. The approach of breaking off the search above the base level of the pyramid to avoid errors caused by holes actually increases the overall performance, but with each level the resulting reflection is increasingly displaced, resulting in a different set of clearly visible errors [Figure 4.3 *(d)*]. Overall, the extension to the rendering system using a ray hierarchy data structure in the form described in this paper can not be said to improve the performance of the underlying rendering technique without any fundamental changes. Additional discussion about the performance can be found in Chapter 6.

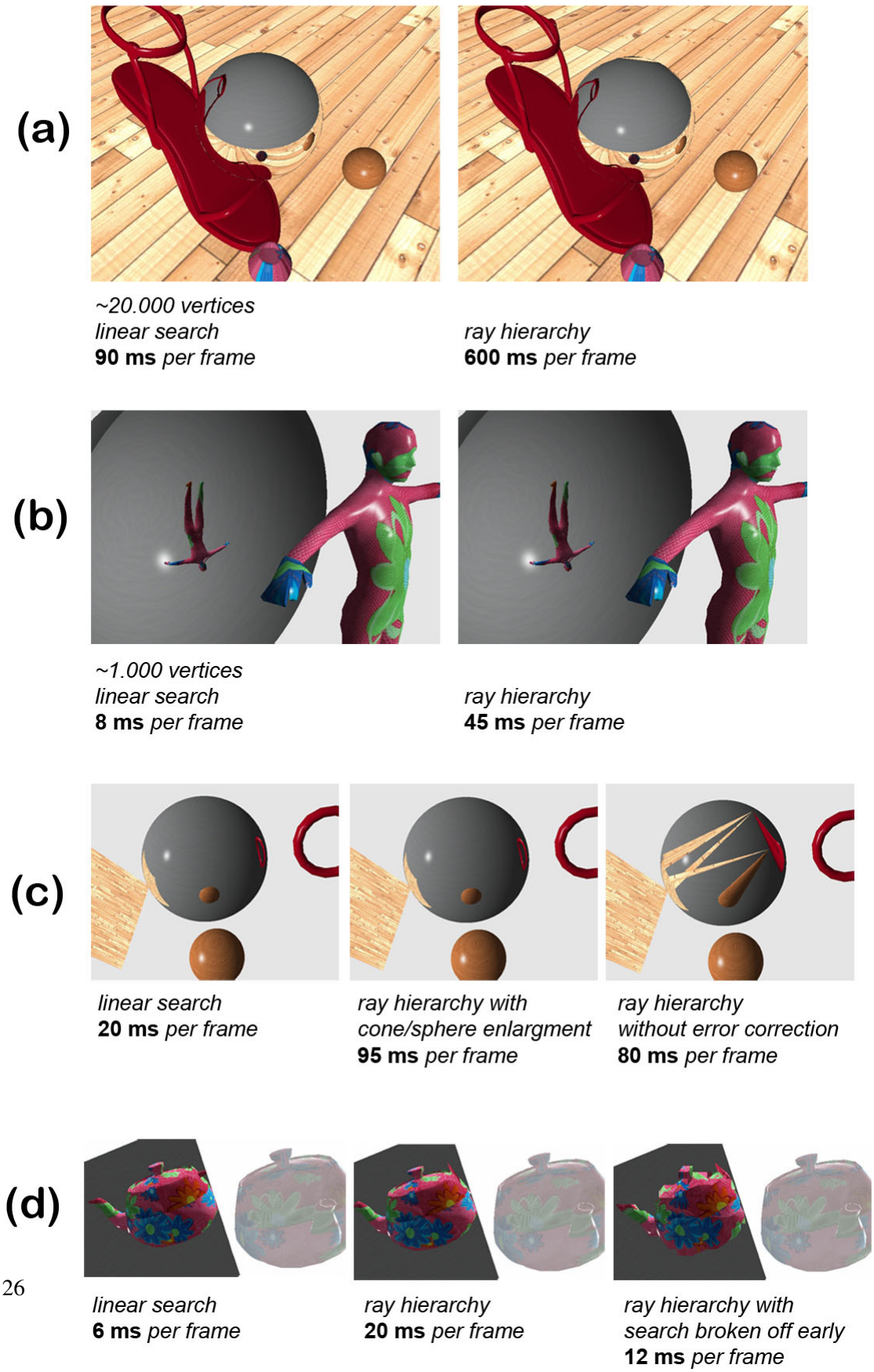

Figure 4.3: Previous examples rendered with hierarchical search. *(a) and (b):* example scenes. *(c):* example for the error when vertices are between cones. *(d):* example for the error when search is broken off prematurely.

CHAPTER 5

Adaptive Reflected Geometry Tessellation

This chapter adresses a problem inherent to rendering reflections on a curved mirror using the graphics hardware for rasterization. The error stems from the fact that the world geometries' vertices' exact reflection points are found, but the graphics hardware then interpolates linearly between those reflection points. The interpolated line should show an appropriate curvature since it represents the reflection of an edge on the curved reflector's surface. Geometry tessellation is used for the purpose of lessening the error between what the reflection of a world geometry looks like and what it should look like.

5.1 Implementation

The geometry's primitives are refined by inserting new vertices along their edges and connecting them to form a new set of primitives with the same outline as the old primitive. When the geometry is reflected off the surface of a curved reflector, the reflection points for both old and new vertices are found on the reflector surface. The edges are then split up into line segments that connect the newly inserted and reflected vertices, resulting in a curved polygon in the place of the edge.

This form of tessellation is computationally demanding, since the amount of reflection point searches to be performed can quickly increase, and therefore some control mechanism is desirable to limit tessellation only to those parts of the geometry that require it. This chapter examines the possibility of defining the *reflection error* of a reflected primitive as the difference between its accuracy regarding the underlying reflector's curvature before and after the tessellation. If the reflection error of the primitive lies above a certain threshold, it is tessellated further and the errors are calculated again, until the errors lie below the threshold or a maximum number of tessellations have been performed. This results in the number of vertices only being increased in parts of the geometry with large difference towards the underlying reflector's curvature.

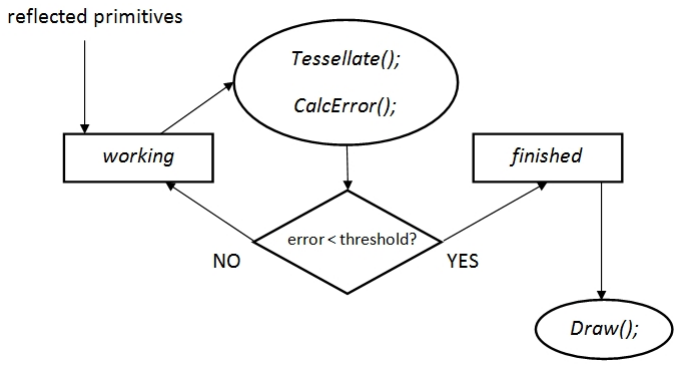

Figure 5.1: A functional overview over the tessellation loop using the two separate buffer objects. Primitives are either stored in the working buffer for further tessellation or appended to the finished buffer for being drawn.

Finally, the stream of primitives of a scene's geometry is split up and stored in two separate buffer objects on the graphics hardware: The *working buffer*, which holds all the primitives that are still being considered for tessellation, and the *finished buffer*, which stores the primitives that are not to be tessellated anymore. This enhancement ensures that the tessellation operation only checks the vertices in the working buffer and does not redundantly attempt to operate on the finalized primitives, which are guaranteed to stay the same, repeatedly.

A loop is established which repeatedly tessellates the primitives in the working buffer, calculates their reflection errors and either sends them back into the working buffer or appends them to the finished buffer. After a maximum number of iterations, any remaining primitives in the working buffer are collected and appended to the finished buffer, and the contents of the finished buffer are finally drawn. The result is the image of the reflection on the curved reflector similar to the previous approach, except that parts of the reflected geometry with too large of an error, according to the curvature of the reflector and some threshold, is tessellated more finely by having some vertices inserted along its edges. A diagram of the tessellation loop's functionality is given in Figure 5.1.

Concretely, the previously examined approaches are modified as follows. The tessellation loop is implemented in the geometry shader stage of the rendering pipeline. The geometry shader receives each of the scene geometry's primitives, which in this case are triangles, as input and outputs the triangle either unmodified or tessellates it according to a tessellation rule. The tessellation rule used here is the simplest case in which one triangle is split into four equally large ones along the median points of its edges. A visualization of the rule is shown in Figure 5.2. If a triangle is tessellated, new points are created as linear interpolation of its vertices, and the reflections are found for the newly-created points.

The reflection error is calculated for each triangle's reflection. For each edge, the reflection error is given by the dot product of the surface normal at the median point between the two reflection points, and the bisector vector between the viewing ray and the reflection ray. The reflection ray is the ray from the median point between the two reflection points towards the

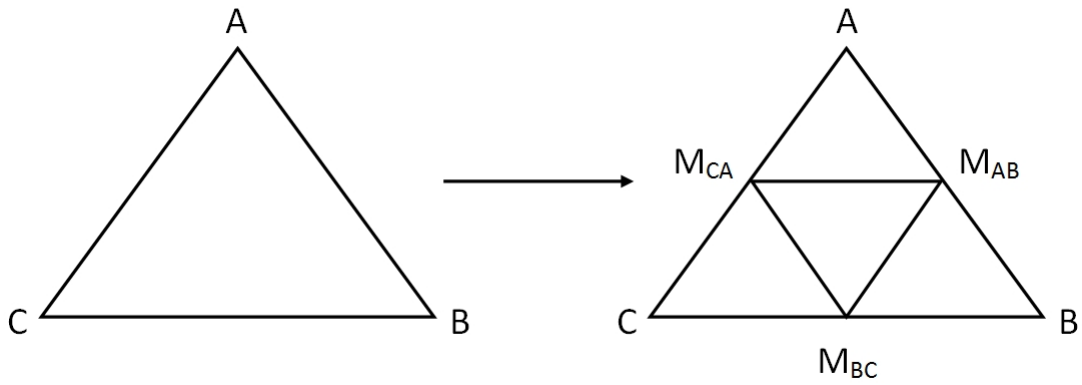

Figure 5.2: The tessellation rule used to tessellate a triangle in the tessellation loop.

median point between their corresponding world vertices. Formally for the two reflection points R_A and R_B of an edge:

$$error = 1 - \frac{N_{R_A R_B} \cdot b_{vr} + 1}{2}$$

where $N_{R_A R_B}$ is the normal at the linearly interpolated median point between R_A and R_B on the reflector surface, v is the viewing ray direction from the median point to the camera position, r is the direction from the median point to the median between the original world vertices A and B, and b_{vr} is the normalized bisector between the two. The dot product is then normalized to lie inside $(0, 1)$, where 0 means no error.

For the error of a triangle, the maximum error of its three edges is taken. After the error has been established for the triangle, it is compared with a global threshold. If the error is smaller then the threshold, the triangle is appended to the finished buffer. If the error is above the threshold, the triangle is outputted to the working buffer. The concrete value for error threshold parameter is dependant on the shape of the reflector. For a ”smooth” object, such as a sphere, the error value is usually very small, around 10^{-3}, whereas for a reflector with high curvature, such as an ellipsoid reflector, the error values may be higher.

Each iteration of the tessellation loop receives the working buffer as input. The loop is repeated until either the working buffer is empty or a set maximum number of iterations has passed, after which any remaining triangles in the working buffer are appended to the finished buffer. Finally, the contents of the finished buffer are drawn.

5.2 Result

The final result is a system which tessellates reflected geometry in order to decrease the visual error resulting from the linear interpolation between the reflected vertices. It does so for each triangle based on a reflection error given by the difference between the linear interpolation and the accurate reflection point of the vertex corresponding to the linear interpolation. By doing

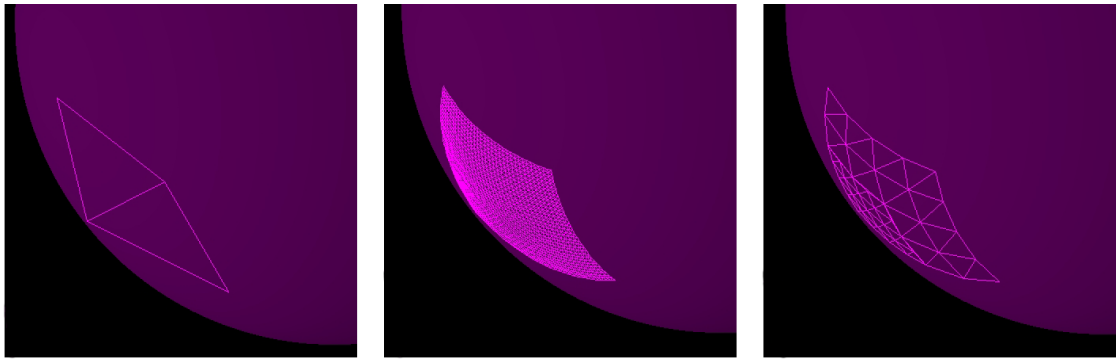

Figure 5.3: An example for a reflection being tessellated. *Left:* A quad consisting of two triangles. *Center:* The same quad fully tessellated with five iterations. *Right:* The same tessellation being performed using adaptive tessellation.

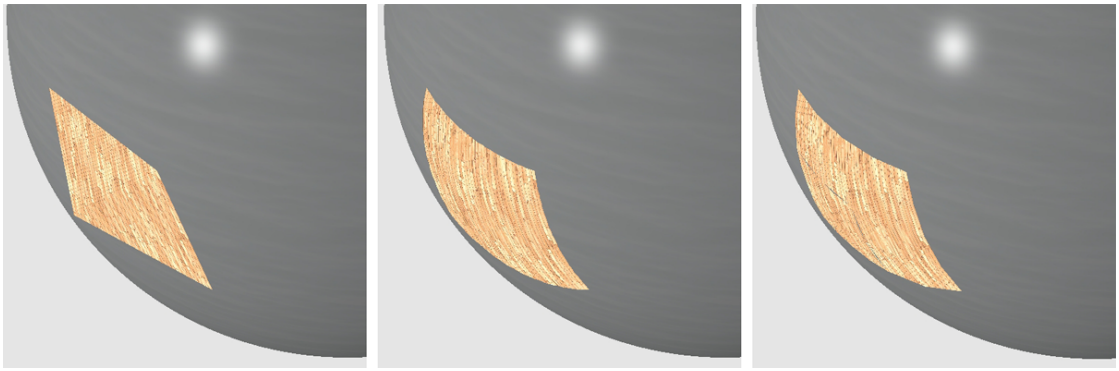

Figure 5.4: The same scene as in Figure 5.3.

so, the system avoids tessellating potentially large numbers of triangles for which almost no increase in visual quality is gained, but still retains the advantages of tessellation for triangles where the error visibly decreases. An example for geometry tessellated in this way is given in Figure 5.3. Figure 5.4 shows the same example with color and shading. It can be seen that the full tessellation and adaptive tessellation result in similar visual quality.

The performance compared to full tessellation increases with the number of tessellations not performed depending on the chosen error threshold, since expensive tessellation steps are skipped and the performance impact of the error calculation is negligible. However, some number of tessellation steps with little impact on quality are still performed, since the error may be high in cases where e.g. the result in the final image would only differ by one pixel. An extension which pre-calculates the necessity for tessellation on a per-object basis or incorporates the pixel size of the projection into the calculation in order to recognize such cases can be imagined. Furthermore, the tessellation rule used here causes holes in previously closed surfaces in locations where a tessellated triangle and an untessellated triangle meet, which would be an anchor point for considering different tessellation rules. Further discussion on the performance

and future work can be found in Chapter 6.

CHAPTER 6

Results Discussion

6.1 Linear vs. Hierarchical Search

In order to illustrate the difference in performance between the linear screen space search based rendering system and the modification using the reflection ray hierarchy, the exact same scene also used in Figure 5.3 has been rendered as an example using both rendering approaches and using different levels of geometry detail. The count of vertices used for the world geometry has been increased steadily while keeping the shape of the geometry the same and the average time in milliseconds required to render a frame has been observed for both techniques. All the rendering has been performed under the same circumstances on a PC with an *Intel Core2Duo* CPU and a *Nvidia GeForce GTX 260* graphics card. A graphical representation of the resulting data can be seen in Figures 6.1 and 6.2.

Figure 6.1 shows the performance of the scene rendered from the same viewpoint as used in the example. For further comparison, Figure 6.2 shows the data when the scene is arranged in a way particularly suitable for the hierarchical search, i.e. the reflection covers the entire viewport and the linear search has to perform the maximal number of search steps. It can be seen that in this case, the two search methods perform approximately similarly. These examples go to show that the search using the reflection ray hierarchy generally overall needs more time to find the reflection point of a vertex than the linear search-based approach. This is solidified by the fact that the example scene used here represents a relatively ”good” case for the ray hierarchy search. All vertices have their reflection on the reflector surface of the ”smooth” sphere, which already lessens the chance of vertices to fall into the empty space between cones and the search step therefore having to be performed again. Additionally, no vertices are hidden by the reflector, in which case the reflection point would not be on the visible side of the reflector and the vertex would be guaranteed to fall outside of any cone. The approach using the reflection ray hierarchy also may produce noticable visible errors in the final image, especially at grazing reflection angles and at the edge of a reflector, depending on the parameters chosen concerning the elimination of invalid search results.

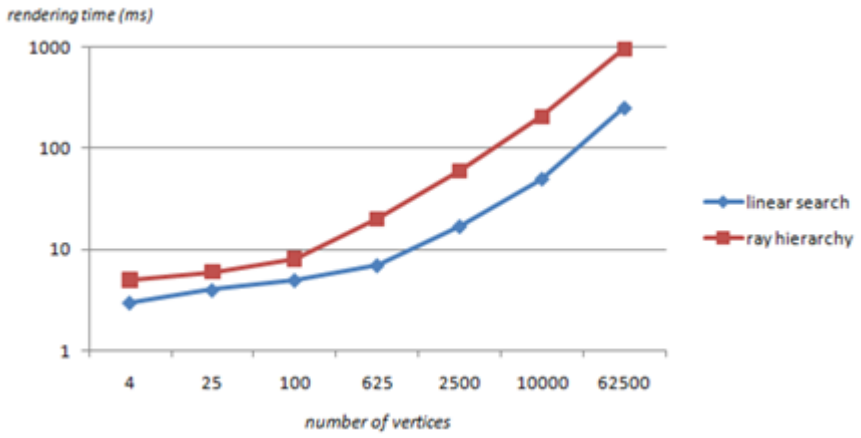

Figure 6.1: A general example scene rendered with both reflection point search approaches, with the number of vertices used in the scene being increased. The resulting times needed to render a frame are measured in milliseconds (ms).

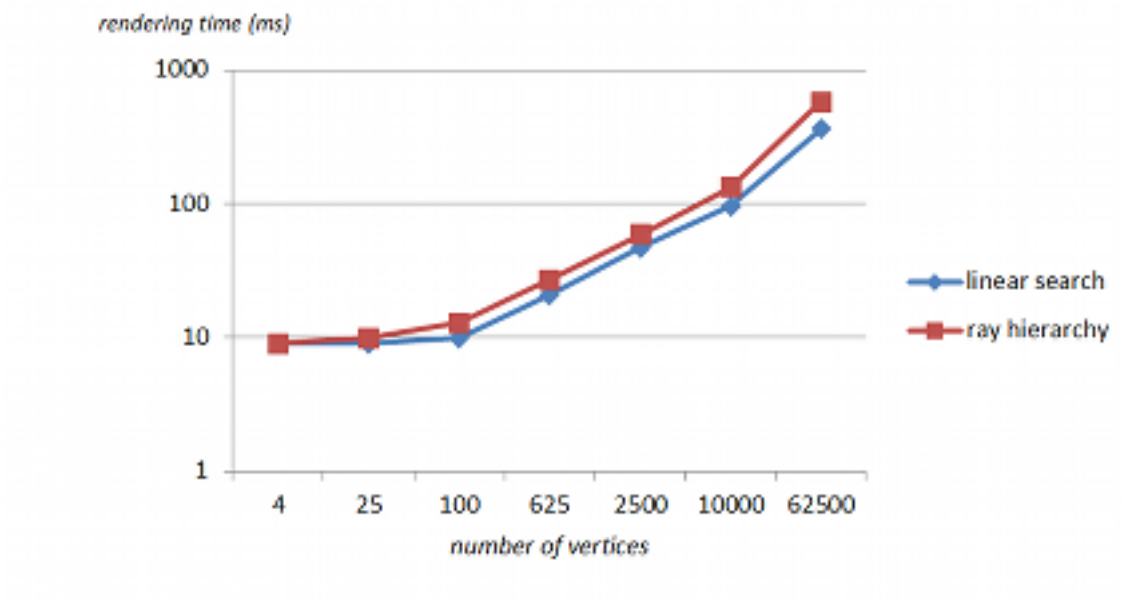

Figure 6.2: The scene rendered here is particularly suitable for the ray hierarchy search.

The main cause for the slower performance is found in the operations used in a search step. When traversing the refletion ray hierarchy, a cone-sphere intersection test has to be performed in each step of the search. Although different routines for performing this test in a way suitable for graphics hardware have been implemented, the operation has always proven to be computationally much more expensive than the simple dot product calculation done in the linear search.

Although this performance impact may have been offset by possibly requiring much fewer search steps for the result, the reflection ray hierarchy search has shown not necessarily always to terminate in a lower number of steps than the linear search. Especially in the case in which the linear search carries the texel location of the search result over to the next frame for use as a starting point in order to utilize frame-to-frame coherence in interactive systems, if the camera stands still, the linear search starts on the correct point and returns its result in one single search step. Similarly, if the camera is moved only slightly, the result is returned in a low number of steps. Such a frame-to-frame optimization is not possible using the ray hierarchy. The search always has to visit each level of the hierarchy at least once and the minimum number of required search steps is at least equal to, but usually higher than, the number of levels in the hierarchy pyramid. Furthermore, since the linear search traverses the image pixel-by-pixel, cases in which the projection of the reflector is very small, but the final image is very large, can be imagined in which the linear search also inherently requires fewer steps.

In addition, the data structure used in the reflection ray hierarchy raises the problem of not covering space uniformly between reflection vectors. The cones, which are used to group reflection rays, may overlap in some areas, causing the search to branch off and examine each of the cones, and other areas may be outside of any cone, an effect that is combatted by repeating the search step in such a case using enlarged cones and spheres. In relation to that, the system has no way of easily differentiating between space that lies between cones but is still valid for reflection, and space that lies outside of the cone hierarchy because it does not factor into the reflection, such as vertices hidden behind the projection of the reflector. This causes more unnecessary search steps compared to the linear search, which recognizes such cases.

In summary, while the searching algorithm based on the reflection ray hierarchy is capable of producing the same results as the linear search, it usually takes more time to do so and can introduce slight visual errors.

6.2 Adaptive vs. Full Tessellation

To illustrate the difference in performance made by using the previously examined system of adaptive tessellation compared to full tessellation, again the same scene from Figure 5.3 has been rendered under the same circumstances, this time varying in the maximum number of tessellation iterations to be performed. The linear screen space search algorithm has been used for finding the reflection points, although the relative difference in performance is the same when using the reflection ray hierarchy approach, since for each newly inserted vertex the search is simply performed one additional time. The threshold for the reflection error used was 0.001. The resulting performance is displayed in Figure 6.3.

This example shows how the number of tessellation steps is effectively limited by the calculated reflection error on a per-triangle basis. Furthermore, if the tessellation level is set very

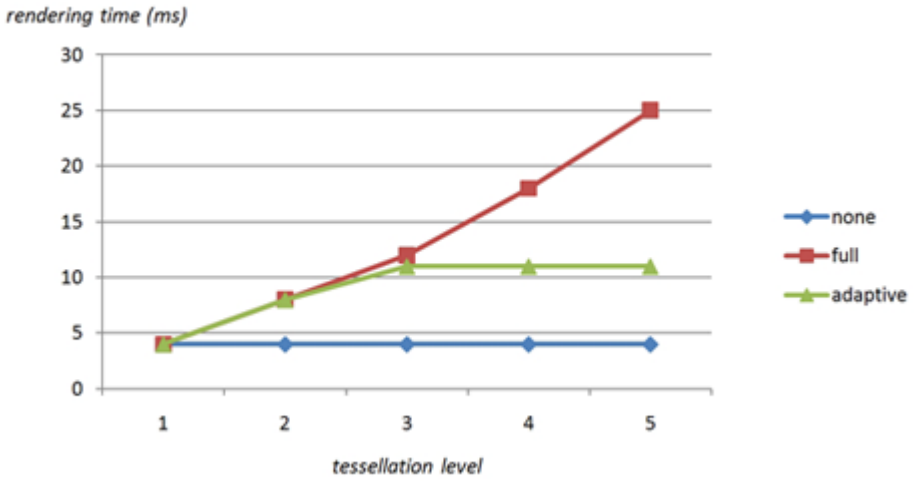

Figure 6.3: The time it takes to render a tessellated reflection, using different maximum numbers of tessellation iterations, without tessellation, with full tessellation and with adaptive tessellation.

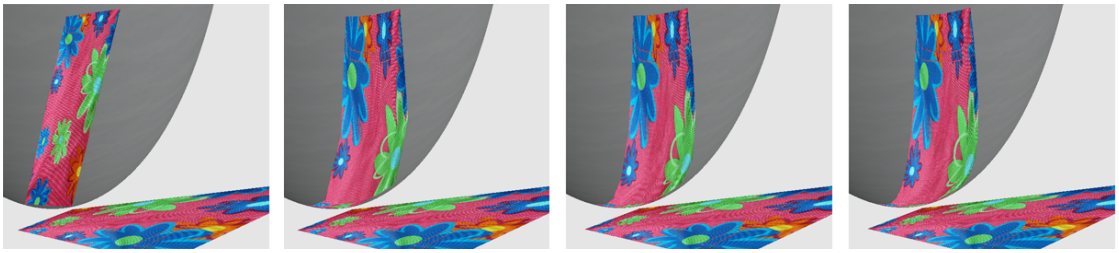

Figure 6.4: *Left to right*: A reflection with the tessellation levels 0, 2, 3 and 4. After level 3, there is almost no visible difference.

high, in this case tessellation level three, the approach does not perform unnecessary tessellation steps. In this example tessellation level two is never exceeded. The resulting image looks practically the same as if it had been tessellated much more often (see Figure 6.4), but comes at a much lower computational cost.

However, some unnecessary tessellation is still performed by this system. For instance, since the error computation does not factor in the actual size of the reflection in screen space, the case in which the reflector is very small may still cause a number of tessellation steps even though the difference in the finally resulting image may be only in singular pixels. In addition, the requirements for the error threshold vary depending on the shape of the reflector. While the value used in this example may prove to work well for the evenly curved mirror sphere, the same threshold may result in undesirable tessellation behavior for a reflector with very strong curvature. As a future work, a method to calculate the tessellation error as a function of the reflector could be found. Finally, the pattern used for tessellation in this system, in which the

Figure 6.5: Extreme example for the hole between two differently tessellated triangles in the reflection of a plane.

entire triangle is tessellated based on the biggest error of its edges, results in holes appearing in the reflection where a tessellated triangle and an untessellated triangle meet. The size of the hole relates to the difference in reflection error of the two neighbouring triangles. Figure 6.5 shows an extreme example for such an error, in which the edges of an untessellated and a tessellated triangle meet. However, the error is usually not this severe since geometry generally consists of many more smaller triangles. The camera also needs to be at the special distance from the geometry and the reflector such that different parts of the geometry get tessellated at different levels for the error to occur. But, since it still does happen, this problem could be alleviated by utilizing different tessellation patterns which respect the location and alignment of the erroneous edges with consideration towards its neighbours.

In summary, although the approach for adaptive tessellation examined here is relatively basic and suffers from the aforementioned problems, it is already in this state capable of providing a massive increase in visual quality for the results of the previously used rendering systems at managable cost in performance, and it serves as an anchor point for future consideration in any multi-pass rendering approach that operates in a similar fashion.

CHAPTER 7

Future Outlook and Conclusion

7.1 Future Outlook

The multi-pass rendering approach that uses the linear screen space search examined in this paper is capable of rendering reflections on curved reflectors at interactive framerates for geometry made up by some tens of thousands of vertices. This may already make it feasible for use in some commercial interactive systems, examples such as simple computer games or mobile games come to mind. However, it is also an interesting starting point for further extensions to increase the range of applications of the system. First of all, since the approach is designed with reflectors in mind that have only convex or only concave curvature, one such extension would make the system applicable to reflectors with mixed curvature. A possible solution to this problem could include a step in which the reflector is segmented into multiple separate reflectors, all of which show uniform curvature, and the reflection being calculated for each reflector individually. The second limitation of the system, which would quickly become evident in such an approach, is that it does not handle self-reflection or reflections of other mirrors. Especially concave reflectors often show self-reflection, for example, in the concave reflection in Figure 3.2 the rim of the mirror should be reflected and visible on its inside, which necessitates separate considerations concerning self-reflection. The mirroring of other mirrors poses a similar problem, in which several layers of reflections inside of reflections would have to be calculated. Overall, much potential still lies in this approach.

The reflection ray hierarchy based search approach has proven to be inferior to the linear search in the way it is implemented in this paper. Some changes to be considered include the use of a different data structure to both make the testing operation in a search step more efficient and to alleviate the problems caused by the current data structure's uneven coverage of the search space. Additionally, the handling of erroneous search results caused by these problems might possibly be improved by using different methods, such as basing the cone and sphere extensions on some property of the world vertex or choosing the child cone by a different approach.

Finally, the adaptive tessellation system examined in this paper provides a powerful tool of increasing the quality of the final rendered reflection while keeping the additional computational

cost relatively low. Future work includes looking into various different methods and patterns for tessellation with the goal of finding the optimal ones for various types of geometry. This goes together with experimenting with different error metrics, such as an error calculated including the size of the reflector's projection, in order to eliminate even more redundant tessellation steps not recognized by the current approach. Other enhancements include calculating the error threshold based on the underlying reflector's geometric properties and computing an object's eligibility for tessellation in a pre-processing step.

7.2 Conclusion

In this paper a method for rendering an accurate reflection on the surface of a curved reflector in real-time has been examined. The method is based around drawing the reflection in a separate rendering pass, in which first the image space reflection position of each vertex of a reflected object is found, and then the reflected geometry is drawn by the graphics hardware by triangulating these new vertices. The searching algorithm for one reflection point operates in screen space and performs its search linearly beginning from a starting point. The method can provide interactive framerate for scenes containing some tens of thousands of vertices.

Afterwards, a possible modification to the system has been considered, which was aimed at improving the time needed for the search for a reflection point to terminate. In this approach, a different data structure is used which can be grouped together using a set of cones and spheres to represent the bounding volume of a set of reflection vectors, which are grouped together to form a hierarchy. The modified searching algorithm for one reflection point traverses the pyramid and finds the result at the bottom level. This modification has proven not to increase performance over the linear screen space search. the main reason for this is that it does not necessarily need fewer steps to terminate, but each individual step is comparatively more expensive in performance, and, in addition, it may introduce errors in the final image.

Following that, a technique for tessellating the reflection geometry in order to bring its curvature closer to the curvature of the underlying reflector has been examined. This technique tessellates the geometry adaptively according to an error metric, which is based on the difference in distance resulting from a tessellation step. As a result, a large number of tessellation steps are skipped if they do not cause a noticable effect in the final image.

Finally, some results have been discussed and ideas for future work in this field have been given, showing that the systems examined in this paper may well serve as anchor points for future considerations extending their applicability.

Bibliography

[1] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[2] Paul J. Diefenbach and Norman I. Badlert. Multi-pass pipeline rendering: Realism for dynamic environments. In *Symposium on Interactive 3D Graphics*, pages 59–70, 1997.

[3] Pau Estalella, Ignacio Martin, George Drettakis, and Dani Tost. A gpu-driven algorithm for accurate interactive reflections on curved objects. In *Proceedings of the 17th Eurographics conference on Rendering Techniques*, EGSR'06, pages 313–318, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[4] Pau Estalella, Ignacio Martin, George Drettakis, Dani Tost, Olivier Devillers, and Frederic Cazals. Cazals f.: Accurate interactive specular reflections on curved objects. In *In Proc. of VMV 2005*, 2005.

[5] Functional Mirrors in Games . http://www.giantbomb.com/functional-mirrors/3015-4618/. Last accessed: 07.02.2013.

[6] Intersection of a Sphere and a Cone. http://www.geometrictools.com/documentation/intersectionspherecone.pdf. Last accessed: 24.02.2013.

[7] Mirrors . https://scs.senecac.on.ca/ gam670/pages/content/mirro.html. Last accessed: 07.02.2013.

[8] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 333–342, New York, NY, USA, 1998. ACM.

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

[10] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 703–712, New York, NY, USA, 2002. ACM.

[11] David Roger, Ulf Assarsson, and Nicolas Holzschuch. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, EGSR'07, pages 99–110, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[12] David Roger and Nicolas Holzschuch. Accurate Specular Reflections in Real-Time. *Computer Graphics Forum*, 25(3):293 – 302, September 2006.

[13] The Story of Reflection Mapping. http://www.pauldebevec.com/reflectionmapping/. Last accessed: 06.02.2013.

[14] Transforming Normals. http://www.oocities.org/vmelkon/transformingnormals.html. Last accessed: 21.02.2013.

[15] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[16] Wikipedia: Ray Tracing. http://en.wikipedia.org/wiki/ray_tracing_%28graphics%29. Last accessed: 20.02.2013.