Real-time Reflections on Curved Objects Using Layered Depth Textures

Cheng Zhang
The Ohio State University
zhangche@cse.ohio-state.edu

Hsien-Hsi Hsieh
National Dong Hwa University
hsi@game.csie.ndhu.edu.tw

Han-Wei Shen
The Ohio State University
hwshen@cse.ohio-state.edu

Figure 1. Reflections on curved reflective objects using layered-depth textures.

Abstract

This paper introduces a novel approach for real-time reflection on curved objects. In our algorithm, we first determine the regions on the reflectors where reflections are likely to exist. To reduce the cost of rendering the reflections in those areas, for each reflected object, a local cube map, called layered depth texture, is constructed. At run time, a fragment program is used to search the intersection point of the reflective ray from the reflector with the layered depth textures. For a convex reflected object, a single-layered cube map is sufficient to ensure geometrically correct reflection results. For a concave object, however, a multi-layered cube map is needed, which can be generated using the depth-peeling technique. Our per fragment based search algorithm for reflection rendering does not depend on the geometry complexity of the reflected objects. In addition, the construction of the layered depth textures does not depend on the viewpoint, so it only needs to be done once. Because the cost of our algorithm is independent of the geometry complexity of reflected objects at run time, real-time reflection computation for complex scenes can be achieved.

1. Introduction

The traditional methods for rendering reflections include ray tracing and environment mapping. Ray tracing images are typically of high quality with more accurate reflections. However, it requires expensive computation, and thus is difficult to be used for real-time applications. Environment mapping, on the other hand, is a fast approach at a cost of lower accuracy. Because the position of the reflector is not considered, nearby objects cannot be reflected correctly and dynamic scenes are difficult to handle. Other than these two methods, there also exist some reflection rendering techniques that employ the virtual object concept. Some of the methods are based on Fermat's principle, while others are hybrids of two or three methods combined together to achieve desired goals. Except the less accurate environment mapping techniques, for most of the existing algorithms, the cost to render reflections increases as the geometry of the reflected objects becomes more complex. Even with simple reflectors, the speed of reflection rendering is largely limited by the number of polygons in the reflected objects.

Many researchers have previously attempted to improve the speed of reflection rendering. Ofek and Rappoport [18] used explosion maps to quickly identify the reflection region on the reflector without exhaustive search. The main

drawback of this approach is, however, if there are several reflectors in the scene, or when the viewpoint is changing, the explosion map needs to be re-computed. Another drawback of their approach is that inaccurate mapped vertices are often found in the explosion map thus the reflection points they computed can be incorrect. Some researchers tried to partially solve the problem by using several methods combined together. Chen and Arvo in [5] combined the ideas of virtual object and ray tracing. They avoided going through all the geometry by first using a sparse set of reflection points, and then applying perturbation to those points to interpolate the nearby points. The main limitation of this algorithm is that the reflectors need to be differential surfaces. In addition, their approach may not work for all the cases when the reflector is concave. Li *et al.* in [14] used ray tracing with geometry fields to render reflections. A geometry field is a combination of a light field with a geometry image [12]. Given a surface, its geometry field is represented as a map that stores all ray/surface intersection information. If an incident ray is given, the geometry field returns the texture coordinates of the first intersection point. With their algorithm, conventional intersection testing is replaced by indexing a geometry field, which does not depend on the complexity of the objects in the scene. This method exhibits inherited limitations from geometry images since the construction process of a geometry field includes the generations of a geometry image and a light field. The construction of a geometry image involves parameterizing a given surface onto a planar domain. For those models with high genus or long extremities, forcing the whole surface to map into a plane can introduce distortions even with conformal parametrization. Moreover, this preprocess itself is expensive and complex.

In this paper, we introduce a novel approach to achieve real-time reflections using graphics hardware. In our algorithm, instead of performing explicit intersection tests or virtual objects rendering and clipping, reflections are rendered on the curved reflectors using cube map-like textures, called layered depth textures, generated from each reflected object. The layered depth texture contains images orthographically rendered from the six faces of a cube surrounding the reflected object. Both the color and depth values are saved in the texture, which only needs to be constructed once. Given an arbitrary camera view, for each reflected ray from a fragment on the reflector, if it intersects with the bounding box of the reflected object, we search the layered depth texture based on the ray marching positions and the depths stored in the texture to locate the intersection point and retrieve the reflection color. Our algorithm can handle both convex and concave objects, and will find the correct intersection point efficiently. For a convex object, a single-layered cube map is sufficient for correct determination of the intersection point. For concave objects, some points can be only visible from particular view points but not from any of the six orthographic views. To handle this case, a multi-layered cube map is produced.

Our method has several advantages. First, the run-time performance of reflection rendering is independent of the geometry complexity of the reflected objects. The size of the layered depth texture grows only linearly with the depth complexity of the reflected object if it is concave. Second, this cube map representation does not depend on the viewpoint of the camera at run time and only needs to be rendered once. Third, since the layered depth texture is created for each reflected objects, our approach overcomes the limitation of environment mapping such as parallax and touching, and can easily handle dynamic scenes. To sum up, because the cost of our algorithm is independent of the geometry complexity of the reflected objects, real-time reflection computation for complex scenes can be achieved using programmable GPUs.

Our paper is organized as follows. We first briefly review the previous work in section 2. In section 3, we describe our algorithm in detail. In section 4, we present experimental results using different scenes. We conclude and discuss future directions of our research in section 5.

2. Previous work

Reflection rendering has been an active research topic in computer graphics for decades. The methods of computing reflection can be classified into two main categories: ray tracing [10] and environment mapping [1]. Ray tracing is a traditional technique for rendering accurate reflections. Because of its high computational cost, full ray tracing is usually not for interactive applications despite the fact that much research has been conducted to accelerate ray tracing using highly parallel computers [25], [26], [27], or GPUs [3], [22]. Some research addresses how to avoid unnecessary rays, intersection testings, and tracing process so that the algorithm can be accelerated. Carr *et al.* in [4] improved GPU ray tracer by using a threaded bounding volume hierarchy stored as a geometry image min-max MIP map. This scheme efficiently inputs nodes of the hierarchy into the GPU pipeline. By updating the hierarchy at run time, the method can ray trace dynamic geometry and capture the secondary reflections. While effective, their implementation cannot handle sharp edges. Another limitation of their method is that it is unable to sort traversal order and often leads to a huge number of unnecessary intersection tests. As a result, their method is not interactive. Li *et al.* in [14] used ray tracing in the geometry field to achieve real-time reflection. A geometry field records all the ray/surface intersection information. In this setting, the expensive intersection testing in the traditional ray tracing is replaced by indexing the geometry field. The main disad-

vantage of their method is the large storage cost and lengthy preprocessing.

Environment mapping [1], [11] is a popular method to compute reflection for interactive applications. It is based on the assumption that the reflected objects (or scene) are located at an infinite distance. When the assumption is violated, the method leads to inaccurate reflection results.

Much work has been done to improve the original environment mapping algorithm by combining it with various image based rendering techniques. Patow [19] used the Z-buffer to compute a distance map along with the environment map. Both the color and the distance to the center of the reflected object are recorded for each pixel in the environment map, which is used to select the proper pixel inside the environment map. Cabral *et al.* in [2] used radiance environment maps with image-based rendering in a reflection space for interactive viewing of arbitrary objects with an extended interactive IBR algorithms, but still suffers from the accuracy problem. Hakura *et al.* [13] proposed Parameterized Environment Maps (PEM) to solve the problem that the environment maps (EM) cannot capture local reflections. However, the method requires large storage space and longer time to precompute the EMs. Moreover, the transitions among different EMs could lead to discontinuity problem. Martin and Popescu in [16] proposed another technique to combine environment mapping with image based techniques. Their method interpolates among several environment maps to handle the parallax issues at the cost of longer preprocess time. Yu *et al.* in [28] used an environment light-field which contains all the information of a light field but is organized like an environment map. Their method also aims to solve parallax issues at the cost of longer pre-computation time. The region of the reflector is also restricted to areas where the light filed or the environment maps can cover.

In addition to these two main categories, some works make use of Fermat's principle to compute reflection. Mitcell and Hanrahan [17] calculated exact reflection paths from curved surfaces defined by implicit functions. They have to solve a nonlinear system numerically to find the reflection points. Therefore, it's a computationally expensive approach. Following Fermat's principle, Roger and Holzschuch [23] computed the exact reflection position by solving an optimization problem where the gradient of the optical path lengths must be zero. However, the accuracy of their method is only at the vertex level. If the reflected object is not tesselated fine enough, interpolation artifacts will appear. Moreover, the approach only captures the first reflections but not secondary ones on reflectors.

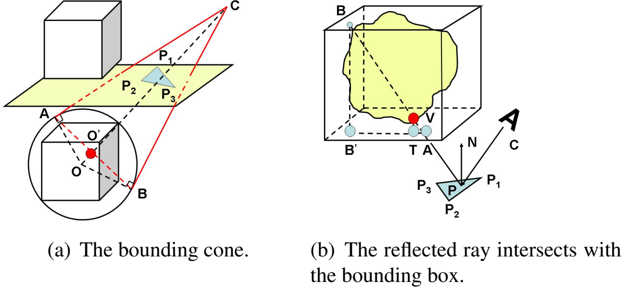
Some researchers have explored the idea of virtual objects proposed in [6] to simulate reflections. Ofek and Rappoport [18] used the virtual object approach to interactively compute single-level reflections on curved objects. The virtual objects are created using a structure called the reflection subdivision and an acceleration scheme, the explosion map. The explosion map needs to be re-computed for each reflector and viewpoint. Their method could obtain incorrect reflection points because an explosion map can often lead to find an incorrect triangle. Chen and Arvo in [5] combined the virtual object idea with ray tracing. They used ray tracing to pre-compute a sparse set of reflection points, then applied perturbation to those points to interpolate the nearby points. However, their approach requires that the reflector be a differential surface. The other limitation is that their approach may not work well for all the cases when the reflector is concave. Estalella *et al.* in [8] [7] also used the concept of virtual objects. For each reflector and each reflected mesh, they create a virtual mesh by using an error function and a search algorithm with a pre-computed cube map for each reflector. The cube maps save the reflector position and normal information, which are needed to search for reflection points at run time. Their method requires the reflectors to be closed objects to fill the reflector cube maps and the objects to be segmented into concave or convex pieces. Moreover, the edges of large polygons are not correctly reflected since they only reflect the endpoints.

In our algorithm, we use layered depth information to find the intersection points between the reflective rays and the reflected objects. The layered depth technique was originally proposed by Shade *et al.* in [24]. In our approach, we extract depth information from different layers of the reflected object, especially for concave objects. Among the various techniques of obtaining depth values, depth peeling proposed by Everitt [9] is more attractive due to its simplicity and robustness. Policarpo *et al.* in [21] proposed an efficient search algorithm to find the best approximated value in their relief mapping. Moreover, they extend their work in [20] to handle non-height-field object in relief mapping. In some sense, our algorithm is similar to relief mapping in terms of finding the intersection points from a precomputed depth map for a given ray.

3. Rendering reflections on curved surfaces

3.1. Overview

Our approach consists of several steps. First, for each reflected object in the scene, we generate a cube map-like texture called layered depth texture in a pre-processing stage. Then, at run time given a particular camera view, the potential reflection regions on the surface of the reflector is first identified. For those regions, we render reflections using a fragment shader to search for the intersection point between the reflected ray and the layered depth textures. Our algorithm handles both convex and concave reflected objects. In the following, we describe our algorithm in detail.



(a) The bounding cone. (b) The reflected ray intersects with the bounding box.

Figure 2. In the left figure, the blue triangle is one of those on the reflective surface and the yellow quad is its expanded plane. C is the camera position. O is the center of the virtual bounding sphere. CA and CB are the lines from C and tangent to the sphere at A and B respectively. All tangent points on the plane perpendicular to line CO forms a circle with its center at $O^{'}$. This circle and the view position C construct a cone. In the right figure, N is the normal at P. A and B are the points that the reflected ray enters and exits the bounding box respectively.

3.2. Determine the potential reflection regions

For a planar reflector, reflection can be calculated in a straightforward way by rendering the virtual object directly. The virtual object is a copy of the reflected object symmetric to the reflection plane computed using a reflection matrix listed in the appendix. We can extend this method to render reflections on curved reflectors when they are modelled as triangular meshes. In this case, a curved reflector can be thought of as being composed of many small planar reflectors. For each small planar reflector in the form of a triangle, if the reflection on this plane overlaps with the extent of the triangle, this triangle has the reflection; otherwise it has not. This test can be efficiently done if we test the triangle against a bounding volume, in the form of a cone, between the camera and the virtual object. To construct such a cone, we first obtain a bounding sphere of the virtual object (see Figure 2(a)); then connect the center of the bounding sphere O with the camera C; all points V_i on the sphere that are tangent to the sphere when connecting to the camera C, form a circle with $O^{'}$ as its center. This circle is the base of the cone. The plane defined by the circle is perpendicular to line CO, and C is the apex of the cone. For examples, in Figure 2(a), A, B are on the sphere, and line CA and CB are tangent to the sphere (i.e., angle $\angle CAO$ and $\angle CBO$ are right angles).

We can change the coordinate system to simplify the cone equation into $x^2 + y^2 = (r - \frac{rz}{h})^2$, where (x, y, z) are the coordinates of a point on the cone surface, r is the radius of the cone base circle, and h is the cone height. In the new coordinate system, $O^{'}$ is the origin, and the vector $\overrightarrow{CO}$ is the positive z axis. Let θ be the angle $\angle ACO$, then $\cos\theta = \frac{|CA|}{|CO|}$ and $|CO^{'}| = \cos\theta \times |CA|$. We can compute $O^{'}$ by the following two equations.

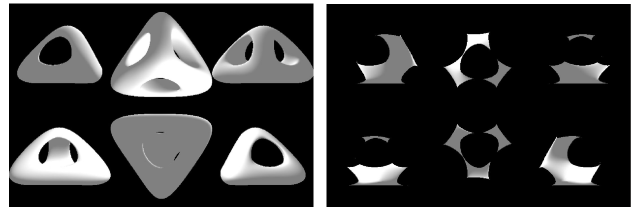
$$\lambda = \frac{|CO| - |CO^{'}|}{|CO|}$$

$$O^{'} = O + \lambda \times (C - O)$$

The expanded plane defined by the triangle can cut through the cone and form an ellipse. Hence the testing becomes to check if the triangle overlaps with the ellipse.

Overall, when a triangle has a reflection, it will satisfy at least one of the following three cases:
1. at least one of the three corners of the triangle is inside the cone;
2. at least one of the three edges of the triangle intersects with the cone;
3. the ellipse is completely inside the triangle.

After identifying the triangles that have reflections on a reflective surface, we will efficiently render the reflections on those triangles with our algorithm described below. The main contribution of our algorithm is to avoid the expensive virtual objects rendering and clipping. This is especially crucial when the number of triangles in the reflection regions is large and the reflected objects are complex. In the following, we present our algorithm in detail.



(a) The first layer depth texture. (b) The second layer depth texture.

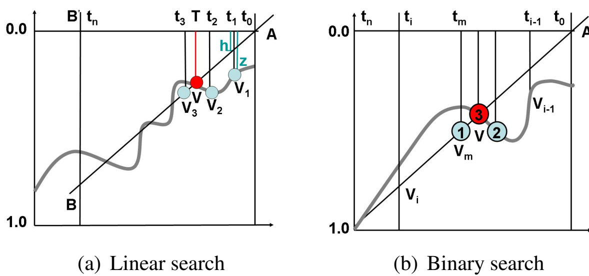
Figure 3. The layered depth textures of the genus3 model.

3.3. Reflection computation

To render reflections, we take an image-space approach to construct a local cube map called layered depth texture for each reflected object. This effectively shifts the work of rendering the reflection geometry to simple texture lookups so that reflection rendering can be accelerated substantially. With orthographic projections, we render each reflected object from six views similar to the construction of a cube

map. For each view, both the color and depth values are read out from the frame buffer and depth buffer to the corresponding cube map face. The colors are stored in the first three channels of 2D $RGB\alpha$ textures, and the depth values are stored in the α channel. The six face textures are packed into one single 2D texture since the graphics hardware usually supports a limited number of texture objects. The packing of six textures are illustrated in Figure 3.

Figure 2(b) illustrates our rendering algorithm. For now, we assume that the reflected objects are convex. We will relax this constraint in the next section and describe how our algorithm handles arbitrary concave objects and construct the layered depth textures. In the figure, $\triangle P_1P_2P_3$ is one of the triangles on a reflective object, P a point on the triangle, N its normal vector, and C the viewpoint. The reflected ray at P is $\overrightarrow{PA}$ computed from its normal and the view vector. A and B are the intersection points that the reflected ray enters and exits the bounding box of the reflected object. V is the point that the reflected ray intersects with the reflected object. The goal of our rendering algorithm is



(a) Linear search (b) Binary search

Figure 4. (a)Linear search is to find the first point inside the object. V_3 is the one, and $h \geq z$ at the corresponding texture coordinate t_3. t_3 and its previous one t_2 are passed as the parameters into the binary search process. (b)Binary search approximates the best value for V within two texture coordinates t_i and t_{i-1}. The numbers here indicate the sequence of approximation.

to, for each fragment P on the reflective triangle, identify the intersection point V based on the depth values stored in the layered depth texture and retrieve its color stored there. Since the computation is per fragment based, we employ a shader program to perform the task of rendering reflections. The process of computing reflections for each fragment can be conceptually described as follows:

- transform point P, its normal vector, and the viewpoint into the local space of the reflected object;
- compute the reflected ray in the local object space;
- compute the two points (A and B in Figure 2(b)) where the reflected ray enters and exits the bounding box;
- search between t_0 and t_n, the texture coordinates for A and $B^{'}$ on the face texture for the intersection point, where $B^{'}$ is the projection of B to the face that the reflected ray first intersects, as shown in figure 2(b).

As illustrated in Figure 4, our search algorithm contains two main steps: (1) an approximate linear search for the first point V_i inside the object; (2) switch to a binary search between t_{i-1} and t_i, where t_i represents the texture coordinates for V_i, and t_{i-1} represents the texture coordinates of the point right before V_i as we are marching the ray. The linear search starts with t_0, the entry point of the ray to the layered depth texture, see Figure 4(a). For each step, we march toward t_n at an equal distance. We compute the height h of the ray and look up the depth z in the corresponding texture for t_i at step i. If h is greater than z, it means the ray point is now inside the object and thus the linear search stops because the intersection point V will be in between the step i-1 and i with texture coordinates t_{i-1} and t_i. We then switch to use a binary search algorithm to identify the intersection point.

As illustrated in Figure 4(b), in the binary search, we calculate the middle point t_m between t_{i-1} and t_i. At t_m, if its height h_m is greater than z_m, discard t_i and continue the binary search between t_{i-1} and t_m. Otherwise, throw away t_{i-1} and continue the binary search between t_m and t_i. This binary search will stop when the error is within a fraction of one texel or it has iterated a pre-set number of steps. Figure 4(b) shows the sequence of search as indicated by the numbers.

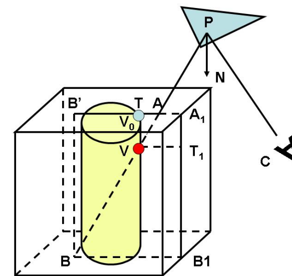


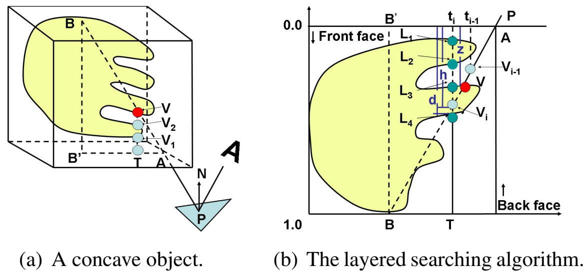
Figure 5. The search process that needs more than one texture.

Our search algorithm is similar to the one which computes ray surface intersection in [21]. However, in our reflection computation, searching only one face texture may not lead to a correct value for the intersection point. This case does not appear in the relief map since the relief map is a small detailed geometry map that is used to enhance a

polygonal surface. Figure 5 shows such a case that one texture is not enough to find the correct value. This is because the depth texture for the face that is first hit by the reflective ray does not capture the actual depth of the intersection point due to the fact that the reflected object, a cylinder, is perpendicular to that face of the layered depth texture.

In Figure 5, point P on the reflective surface should reflect the color value of V on the reflected object. If we only search for V in the top face texture, we will get an incorrect texture value T since all points between V_0 and V are projected onto the same point T in the top face texture. The point V_0 with the smallest depth occludes all other points along the vertical line. If we use the right face texture, however, to perform the search, the correct intersection point can be found because the correct value T_1 for V is stored there. The number of the faces we need to search and compare from the layered depth texture is at most three. The three faces that are selected for the search process is decided by the direction of the reflected ray. By negating the sign of each component of this normalized vector, we can choose the three textures. We can simplify the three-texture searching in our algorithm by setting a threshold. The search process can stop early when h and z are close to each other within the threshold.

The algorithm described in this section is sufficient for convex reflected objects. However, the reflections of concave objects cannot be correctly rendered with a single-layered depth texture. For example, in Figure 6(a), the vertex V is blocked by $V1$ from the bottom view but is also not visible to the right face view. In the next section, we describe how to handle the concave cases.



(a) A concave object. (b) The layered searching algorithm.

Figure 6. The left figure shows a concave object in our approach. The right illustrates that a reflected ray PB intersects the object at V. Texel t_i has 4 depth values at $L1$, $L2$, $L3$, and $L4$. Among them, $L1$ and $L3$ are the front face depths. $L2$ and $L4$ are the back face depths.

3.4. Handle concave objects

The algorithm described above can be extended to handle concave reflected objects. Basically, this is done by: (1) construct a multi-layered depth texture that stores the depth and color information (examples are shown in Figure 3); (2) modify the search algorithm presented above to handle an arbitrary number of layers. This situation is illustrated in Figure 6(b). For the bottom face, the pixel at T contains two layers (at $L4$ and $L2$) with back faces (at $L3$ and $L1$) ignored.

To generate a multi-layered depth texture, we use the depth peeling technique described in [15] and [9] to extract the desired information for each layer. Besides colors and depths are saved and packed into the textures as described previously, we also construct another texture to include surface normals in the first three channels, and the layer count associated with current pixel in the α channel. The layer count indicates how many layers that the current pixel has, which will be used by the search algorithm. We assume that the concave objects are defined as closed surfaces, which means that all front faces are paired with the corresponding back faces. For example, the concave object shown in Figure 6(b) at texel t_i, has two front faces L_1 and L_3 (if we view the object from the top face), and two back faces L_2 and L_4 corresponding to its front faces. The layer count associated with the texel t_i in the cube map texture in this case is two. We define three variables h, z, and d. h is the height computed along the reflected ray $\overrightarrow{PB}$. z is the depth read out from the current face texture in the current layer. d is one minus the back face depth of the current layer. To make efficient use of the texture memory, for each face of our cube map texture, we do not store the back face depths. However, the back face depth can be found from the opposite side of the current cube map face since it is seen as the front face there. With this multi-layered texture setup, our linear search algorithm not only needs to find a proper point inside the object, it should also identify which layer that the point resides when the linear search ends. For each point along the ray with h being the height, there are two cases that can help us distinguish which layer an intersection point may reside: (1) if $h \leq z$, the intersection point is in the current layer; (2) if $h > z$ and $h < d$, the intersection point is in the layer that is greater than the current one. So the layer number increases until the condition (1) is held in some layer or the program stops if all the layers in current texel run out without meeting the criteria, i.e, no intersection is found. Then the search process marches to the next step with a new texel. The pseudo code of the linear search is listed in algorithm 1.

When the linear search ends up at a point inside the reflected object corresponding to a particular layer, the binary search starts. The layer number, and two texture coordinates t_{i-1} and t_i are passed into as parameters. The binary search keeps refining the value between t_{i-1} and t_i in the texture until the intersection point is found.

Algorithm 1 LinearSearch(t_0, t_n, faceId)

1: i=1; j=1; $find$=false; t = t_0; dir=normalize($t_n - t_0$);
2: **while** (**not** $find$ **and** $i \leq linearSearchSteps$) **do**
3: h = computeDepth(t);
4: pixelLayers = getPixelLayer(faceId, t);
5: j=1; subFind=false;
6: **while** (**not** $subFind$ **and** $j \leq pixelLayers$) **do**
7: z=getDepthWithLayer(faceId, t, j);
8: d=getBackfaceDepth(faceId, t, j);
9: **if** ($h < z$) **then**
10: $subFind$=true;
11: **else**
12: **if** ($h \leq d$) **then**
13: layer=j; $find$=true; $subFind$=true;
14: **end if**
15: **end if**
16: j++;
17: **end while**
18: i++; t = t_0 + stepSize$*i*$dir;
19: **end while**

Figure 7. The reflections of the hollow-box touching the reflective sphere.

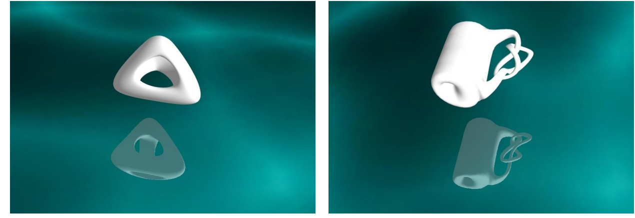
4 Results

This section presents some experimental results using our real-time reflection rendering technique. All experiments are conducted on a windows XP system with a 2.19GH_z Intel processor, 1GB RAM and an NVidia GeForce 7800 GPU. The resolution of our images generated is (768×512).

4.1. Experimental results

The geometry information about the models used in our experiments are listed in Table 1, which includes the model size, the number of the depth layers, and the pre-processing time (in milliseconds) for creating the layered depth textures. The pre-processing speed is dependent on the size and the model's depth complexity.

Table 2 shows the performance of our algorithm with experimental scenes. The second column of the table lists the time (in milliseconds) just to render the reflected objects with OpenGL, which is solely determined by the geometry complexity of the model and unrelated to the performance of our reflection algorithm. Column three and four of the table lists the speed of the different stages in our algorithm. Column three lists the time for rendering the reflector with a global environment mapping but not the reflections of nearby objects. Column four lists the performance of our real-time reflection algorithm for the reflected objects running on the fragment shader. Column five shows the total frame rates, which include rendering the reflector, the reflected objects, the global environment mapping, and the reflections of nearby reflected objects. Alpha blending is used to combine the global and nearby reflections.

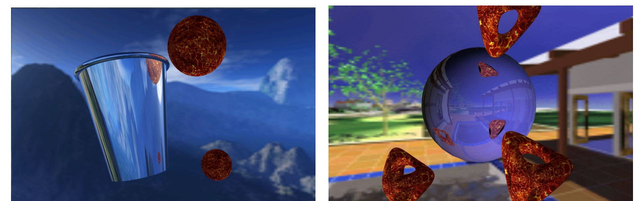


(a) Genus3 and its reflection. (b) Knotty-cup and its reflection

Figure 8. The reflections of two non-zero genus models.

Table 1. Model sizes and the time (in milliseconds) to create the layered depth textures.

Models	Triangular Faces	Layers	Time
Ball	1520	1	24.783
Hollow-box	135	2	46.489
Dancer	49996	2	50.372
Knotty-cup	10768	3	63.289
Genus3	13312	3	63.592
Elephant	25086	3	65.789
Horse	96966	3	71.075
Armadillo	49998	5	95.328
Tyra	49998	5	95.827
Happy-Buddha	129484	5	110.000



(a) Two textured balls and their reflection. (b) Three textured genus3s and their reflection

Figure 9. The reflections of textured models.

Figure 10. Reflections of the armadillo and tyra.

Table 2. Performance (in milliseconds) of our algorithm with various experimental scenes.

Scene	Objects*	Reflector	Reflection	FPS
2 balls	0.195	1.013	1.559	52
Hollow-box	0.154	1.019	1.612	90
Knotty-cup	0.316	1.023	2.131	60
Buddha	4.25	1.027	2.188	55
Armadillo	2.439	1.022	2.337	75
Tyra	1.95	1.021	2.342	75
2 elephants	0.592	1.064	4.324	45
2 horses	5.684	1.024	4.822	40
3 genus3s	0.568	1.013	5.372	40
3 dancers	2.443	1.029	6.261	35

Objects* - The rendering time of reflected objects.

4.2. Comparisons with other reflection methods

While it is difficult to compare our approach with all the other existing reflection rendering methods, there are some notable advantages in our algorithm. For example, we can achieve more accurate reflection images than environment mapping without sacrificing the performance when the reflected objects are close to or even touch the reflector, as shown in Figure 7. In terms of speed, our algorithm outperforms ray tracing due to its simplicity. The performance of our algorithm is also independent of the size of the reflected models and only related to the depth complexity of the layered depth textures. This can be seen from Table 2.

Unlike the explosion map in [18], the construction of the layered depth textures for each reflected object is independent of the viewpoint, so it only needs to be done once. In addition, explosion maps can lead to inaccurate reflections because they do not compute the exact explosion map cells to all mapped vertices. See the discussion section in the original paper.

Compared with the method of ray tracing using the geometry field in [14], we have advantages as follows:

- During the preprocessing, Li *et al.*'s method needs to generate the geometry image, the geometry field, and parametrization etc, so it can take much more time than our algorithm. Even for concave objects, the time of obtaining a multi-layered cube map in our algorithm increases only with the depth complexity of the object, but not its geometry complexity.

- In terms of space consumption, Li *et al.*'s approach requires a large size of storage for the geometry fields to store all ray directions even for simple geometry models. In their paper, all geometry fields have the size of $(24 \times 24 \times 6)^2$, even for a simple sphere. In our approach, we do not need to quantize and store all reflective ray directions. For a simple sphere, for example, only a single layered depth texture is needed. If the resolution is 256, then we only need $(256 \times 256 \times 6)$ for the storage.

- Li *et al.*'s algorithm employs geometry images, which are stored at a fixed resolution. For high genus models, geometry images can often contain errors. Since the depth information of all layers are stored in our algorithm, we can capture those high genus structures correctly. See the reflection images of knotty-cup and happy-Buddha in Figure 1 and Figure 8.

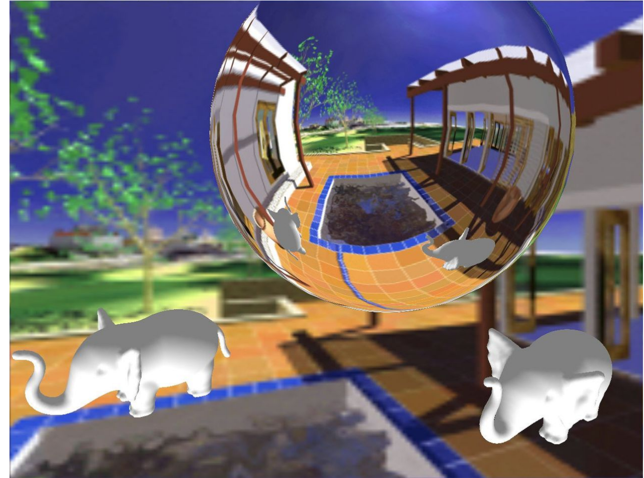


Figure 11. The reflections of two models.

Figure 12. Three dancers and their reflections

5 Conclusions and future work

We have presented a simple and interactive technique for real-time reflection rendering. After first determining the potential reflection areas on the reflectors, we make use of programmable graphics hardware to compute reflections on these regions. To efficiently and correctly find the colors of reflection points, we construct a multi-layered cube map called layered depth texture with a depth peeling technique. We also implement an efficient search algorithm to find the best values of reflection points based on the stored multi-layered cube map, which ensures that the time complexity of our algorithm is independent of the complexity of the reflected objects. Therefore we can enhance the performance and achieve interactive frame rates.

Our algorithm, however, also has some limitations. The current implementation does not yet handle occlusions. For example, the reflection should not appear on the reflective surface if it is blocked by the reflector itself, or other objects in the scene. In addition, the current work only computes the first order reflections but not secondary reflections such as the self reflections or inner reflections. These are all important to improve realism and we are exploring ways of capturing and representing those features.

Appendix

Given a triangle, $P_1P_2P_3$, where $P_1 = (x_1, y_1, z_1), P_2 = (x_2, y_2, z_2), P_3 = (x_3, y_3, z_3)$. Let $\overrightarrow{n}$ be the normal of the plane defined by the triangle. Then $\overrightarrow{n} = (P_1 - P_2) \times (P_3 - P_2)$. Let a, b, c be the three components of $\overrightarrow{n}$, then the plane equation is: $ax + by + cz + d = 0$. The mirror

matrix to compute the virtual object will be:

$$\mathbf{M} = \begin{pmatrix} 1-2a^2 & -2ab & -2ac & -2ad \\ -2ab & 1-2b^2 & -2bc & -2bd \\ -2ac & -2bc & 1-2c^2 & -2cd \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}.$$

References

[1] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[2] B. Cabral, M. Olano, and P. Nemec. Reflection space image based rendering. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 165–170, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[3] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.

[4] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, 2006.

[5] M. Chen and J. Arvo. Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics*, 6(3), 2000.

[6] P. J. Diefenbach and N. I. Badler. Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, 1997.

[7] P. Estalella, I. Martin, G. Drettakis, and D. Tost. A gpu-driven algorithm for accurate interactive reflections on curved objects. In *Rendering Techniques'06 (Proc. of the Eurographics Symposium on Rendering)*, June 2006.

[8] P. Estalella, I. Martin, G. Drettakis, D. Tost, O. Devillers, and F. Cazals. Accurate interactive specular reflections on curved objects. In *Proceedings of Vision Modeling and Visualization*, 2005.

[9] C. Everitt. Interactive order-independent transparency, 2001.

[10] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, Erewhon, NC, 1989.

[11] N. Greene. Environment mapping and other applications of world projections. *IEEE Comput. Graph. Appl.*, 6(11):21–29, 1986.

[12] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002.

[13] Z. S. Hakura, J. M. Snyder, and J. E. Lengyel. Parameterized environment maps. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001.

[14] S. Li, Z. Fan, X. Yin, K. Mueller, A. E. Kaufman, and X. Gu. Real-time reflection using ray tracing with geometry field. In *EG '06: Proceedings of the 2006 EUROGRAPHICS conference*, pages 29–32. Canadian Information Processing Society, 2006.

[15] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9(4):43–55, 1989.

[16] A. Martin and V. Popescu. Reflection morphing. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, page 152, New York, NY, USA, 2004. ACM Press.

[17] D. Mitchell and P. Hanrahan. Illumination from curved reflectors. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1992. ACM Press.

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

[19] G. A. Patow. Accurate reflections through a z-buffered environment map. In *Proceedings of Sociedad Chilena de Ciencias de la Computacin*, New York, NY, USA, 1995. ACM Press.

[20] F. Policarpo and M. M. Oliveira. Relief mapping of non-height-field surface details. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 2006.

[21] F. Policarpo, M. M. Oliveira, and J. L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, 2005.

[22] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[23] D. Roger and N. Holzschuch. Accurate specular reflections in real-time. *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25(3), sep 2006.

[24] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, New York, NY, USA, 1998. ACM Press.

[25] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, 2001.

[26] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 153–164. Blackwell Publishing, 2001.

[27] S. Woop, J. Schmittler, and P. Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[28] J. Yu, J. Yang, and L. McMillan. Real-time reflection mapping with parallax. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 133–138, New York, NY, USA, 2005. ACM Press.