https://doi.org/10.31896/k.25.10
Review
Accepted 2. 12. 2021.

CHRISTIAN CLEMENZ
LEONARD WEYDEMANN

Reflection Techniques in Real-Time Computer Graphics

Reflection Techniques in Real-Time Computer Graphics

ABSTRACT

Reflections have a long history in computer graphics, as they are important for conveying a sense of realism as well as depth and proportion. Their implementations come with a multitude of difficulties, and each solution typically has various trade-offs.

Approaches highly depend on the geometry of the reflective surface since curved reflectors are usually more difficult to portray accurately. Techniques can typically be categorized by whether they work with the actual geometry of the reflected objects or with an image of these objects. For curved surfaces, image-based techniques are usually preferred, whereas for planar surfaces the reflected geometry can be used more easily because of the lack of distortion. With current advances in graphics hardware technology, ray tracing is also becoming more viable for real-time applications. Many modern solutions often combine multiple approaches to form a hybrid technique.

In this paper, we give an overview of the techniques used in computer graphics applications to create real-time reflections. We highlight the trade-offs that have to be dealt with when choosing a particular technique, as well as their ability to produce interreflections. Finally, we describe how contemporary state-of-the-art rendering engines deal with reflections.

Key words: reflections, interreflections, real-time rendering

MSC2010: 51-04, 51p05, 78A05

Tehnike zrcaljenja u Real-Time računalnoj grafici

SAŽETAK

Zrcaljenja imaju dugu povijest primjene u računalnoj grafici zbog njihove važnosti u prenošenju realističnosti prikaza te prikaza dubine i omjera na slikama. Pri implementaciji zrcaljenja dolazimo do raznih teškoća i svako novo rješenje često imaju svoju cijenu.

Pristupi implementacije ovise o geometriji plohe na kojoj leži prikaz, što je ploha zakrivljenija, to je teže postići vjerni prikaz. Tehnike možemo kategorizirati u one koje rade sa stvarnom geometrijom zrcaljenih objekata te one koje rade samo sa slikama objekata. Kod zakrivljenih ploha koriste se tehnike bazirane na slikama, dok se kod ravninskih ploha koristi zrcaljena geometrija jer nema iskrivljenja. Zahvaljujući trenutnom razvoju tehnologije grafičkih hardvera, metoda praćenja zraka (ray tracing) postaje sve isplativija u real-time primjeni. Mnoga moderna rješenja kombiniraju razne pristupe i dolazi do hibridnih tehnika.

U ovom radu dajemo pregled tehnika korištenih u primjeni računalne grafike za postizanje real-time zrcalnih slika. Naglašavamo probleme koji nastaju pri korištenju određene tehnike te njihove mogućnosti u pogledu stvaranja međuzrcaljenja. Naposljetku, opisujemo kako moderni alati za renderiranje rješavaju probleme zrcaljenja.

Ključne riječi: zrcaljenje, međuzrcaljenje, real-time renderiranje

1 Introduction

Reflections have been a research topic in Computer Graphics for over forty years because of the big part they play in depicting realistic scenes. They have a great impact on how we perceive things. For example, mirrors can make small spaces look much larger by giving a sense of depth. They can also convey if a surface is rough or smooth and whether

it is planar or curved. We were made aware of these important properties and the complex topic, when we created a scene that demonstrates the geometry of a C-60 fullerene using multiple mirrors as seen in Figure 1.
Recently, major advances have been made on the topic of real-time reflections. In addition, their field of application grew as well. Besides their typical use in video games, real-time reflections are now also used in architectural visualization and movie production. But older techniques are also still relevant to this day. Depending on the specific application, each technique has its own advantages and disadvantages.
In the following chapters, we give an overview of the current state-of-the-art techniques to provide the reader with an outline of the advantages and drawbacks one needs to consider (Section 2) and we discuss contemporary multipurpose rendering engines and how they deal with reflections (Section 3).

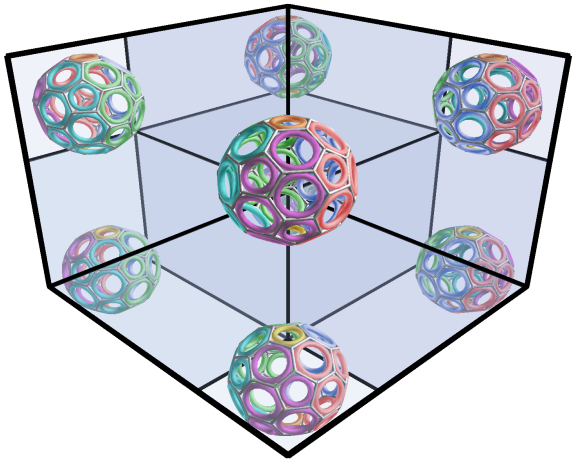

Figure 1: *Interreflections of a single C-60 fullerene created with our implementation of a geometry-based reflection technique. The fullerene is placed in front of three orthogonal mirrors, which are positioned on the XY, XZ and YZ plane respectively.*

2 Techniques

Over the years many different techniques have been developed to create real-time reflections. McReynolds and Blythe [12] categorize them into two groups: object-space and image-space techniques. The former work directly with the geometry while the latter uses textures to create reflections. So, henceforth, we will label these techniques *geometry-based* and *image-based* respectively. Historically, ray tracing was not used in real-time applications because of its long computation time per frame. In recent years, it has become more and more advanced to allow for interactive frame rates. Additionally, the development of new graphics hardware, that has dedicated ray tracing capabilities, has made it suitable for a wider range of real-time applications. Because of this, we include them as a category in our list of techniques. Besides those categories, there are many hybrid techniques that combine multiple approaches to alleviate their individual shortcomings. In this section, we discuss each category in detail, showing examples and considering their advantages and disadvantages.

2.1 Geometry-Based Techniques

McReynolds and Blythe [12] describe geometry-based techniques as approaches that directly transform the geometry of the reflected object. In other words, they create virtual objects that are transformed to represent reflections. This process highly depends on the surface of the reflector.

2.1.1 Planar Surfaces

For planar surfaces, a single affine transformation for each object is enough to describe its reflection, since the reflector's surface normal does not change. This means that it can easily be computed and applied as an additional transformation matrix for example.
Geometry-based techniques need an additional clipping stage, as the virtual object that is created can protrude the plane of reflection or extend beyond its boundary. According to McReynolds and Blythe [12] clipping can easily be done for planar reflectors either by defining custom clipping planes, which the graphics pipeline can use, or by using the stencil buffer to distinguish between pixels that belong to the reflective surface and those that do not. The stencil buffer approach can either be done by rendering the reflector first and then only render the reflected objects inside the stencil or by rendering the reflections first and clearing the image buffer around the stencil afterwards. The second approach can be faster, because the stencil is only used for one clearing operation and not for rendering every individual reflected object. The first approach is better suited for interreflections between multiple reflective surfaces, since the stencil can contain flags that distinguish between different reflectors and the depth of reflection. An example of our implementation using the stencil buffer is shown in Figure 2. We use the stencil to determine where to draw the virtual objects.

2.1.2 Curved Surfaces

In the case of curved reflectors, it gets more complicated. Reflections now also depend on the viewpoint, which can be seen in Figure 3. Therefore, they must now be computed for each vertex individually by finding the correct intersection point of the viewing ray and the reflector sur-

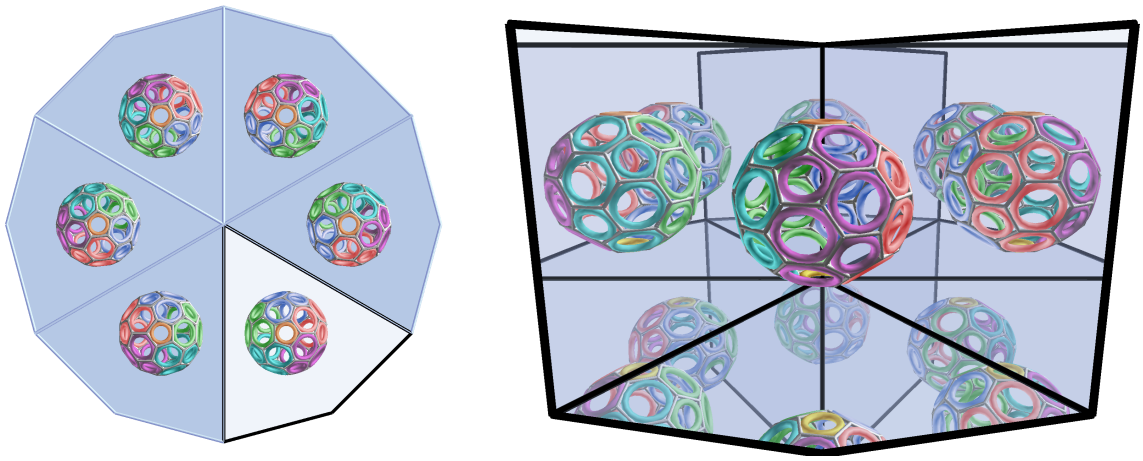

Figure 2: *Our simple reflection setup. We mirror the object for as long as it remains in front of any mirror plane. The left image shows a top-view of the reflections. The final result on the right is created by using a stencil buffer to only render pixels that are inside the mirrors bounds.*

face. McReynolds and Blythe [12] mention that a closed-form solution for finding the reflection point for arbitrary viewpoints, reflector positions, surface shapes, and vertex positions can be very difficult and is usually too complex to generalize.

Ofek and Rappoport [15] proposed a solution for reflections on curved reflectors that creates virtual objects by reflecting each polygon's vertices. They assume that the reflector itself is represented by a polygonal mesh. If this was not he case, they would tesselate the reflector. Each polygon on the reflector divides the space around the reflector into a hidden and a visible cell. Each reflected object is also tesselated depending on the desired resolution of the result. Afterwards, each polygon of the reflected object is reflected. This is done by finding the virtual reflected vertex for each vertex in the polygon. In order to reflect this vertex correctly, Ofek and Rappoport find the polygon on the reflector that is used as the mirror. To prevent the result from looking like a linear approximation, they use the barycentric coordinates of the mirrored vertices inside the cell above the reflector polygon to interpolate between the three tangent planes associated with the reflector polygon. This interpolation is then used as the final plane of reflection for that particular vertex. In order to quickly find out in which cells the vertices are located, Ofek and Rappoport [15] use an *explosion map* as their data structure. Explosion maps are very similar to environment maps, which we will discuss in Section 2.2.1. Instead of color information the map contains polygon IDs to quickly find surface polygons for any given UV coordinate. They claim that their method works best for convex surfaces but it also works for concave surfaces. Surfaces that have both convex and concave areas should be split into separate meshes.

McReynolds and Blythe [12] mention that clipping the virtual objects created by such a method against curved reflectors directly is possible but can be a time consuming operation if the reflector is complex. An alternative would be to use the depth buffer to only render objects with greater depth than the reflector, but this would also render them incorrectly if one virtual object occludes another one.
To summarize, creating reflections using curved reflectors paired with a geometric approach can be very complicated, depending on scene size and complexity. The results are relatively accurate but usually other approaches are preferred for curved reflectors, as we will see in the next section.

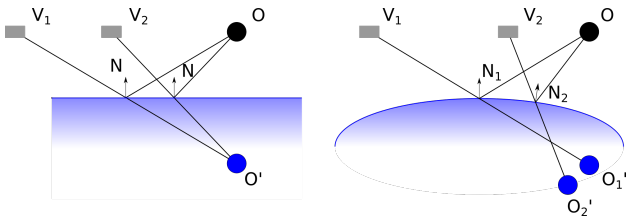

Figure 3: *A comparison of reflection rays on planar and curved surfaces. On the left the object O gets reflected to the same virtual position O′ because the surface normal N does not change. On the right the same object's reflection point varies depending on the viewing position.*

2.2 Image-Based Techniques

As the name implies image-based techniques use images or textures to create reflections. McReynolds and Blythe [12] state that these textures are then used for the reflective surface which is the case for environmental mapping.

Additionally, we also include approaches into this category that use the final or intermediate rendered image itself.

2.2.1 Environment Mapping

An early technique that was developed to create reflections is *environment mapping*, which is also often called *reflection mapping*. The idea is to project the scene onto the surface of a primitive centered around the reflective object. This is done by rendering the scene, viewed from the center point of the reflective object, onto six images forming a cube. These images are mapped onto the primitive using a mapping function that depends on the type of primitive. During the rendering step, another function is needed to retrieve the information from the map.
One of the most popular environment mapping methods is cube mapping and was proposed by Greene [7]. The map is created as described above and uses the cube formed by the image planes directly without re-mapping. The cube can be aligned with the coordinate axes, so that the largest vector component of the reflected viewing direction determines the face that needs to be indexed directly. The texture coordinates are determined with the remaining two components. If the cube is not aligned, the cube faces have to be tested for intersection with the reflected viewing ray. An example of cube map indexing can be seen in Figure 4.

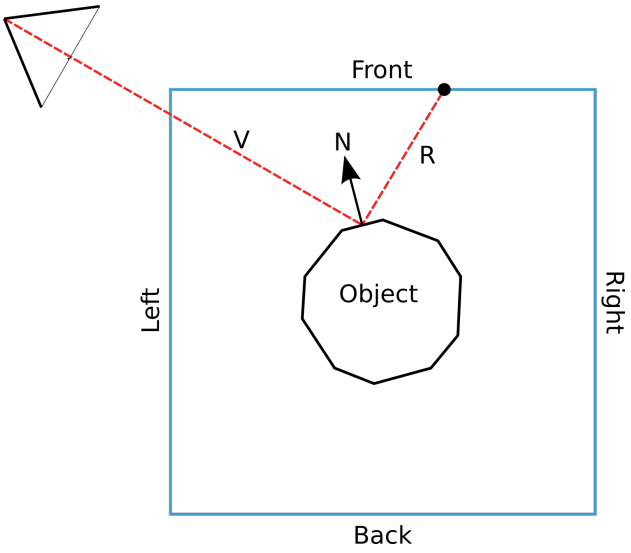

Figure 4: *A top-view of cube map indexing. The viewing direction V is reflected in the object's surface normal N. The reflected direction R determines the cube face and the texture coordinates to use for the final color value.*

Another technique that was proposed very early on by Blinn and Newell [4] is sphere mapping, which uses a sphere as the primitive onto which to map the environment. The key difference is that the image planes get mapped onto a sphere whose surface is then re-mapped to a circular shape inside a 2D texture. This has the advantage that all information is contained in only a single image. However, sphere maps also have drawbacks. Some texture space is wasted since the texture itself is rectangular. But more importantly, they introduce sampling problems. While texture coordinates are interpolated linearly, sphere maps are non-linear. This leads to interpolation artefacts, especially close to the edge of the circular image.
Regardless of which primitive is used, environment mapping is especially useful for curved reflectors, because reflections can be calculated without complex geometrical transformations for each object's vertex in the scene. In some cases, it is also convenient that the maps can be preprocessed if the surroundings or the reflectors are static. On the other hand, if either the surroundings or the position of the reflector, i.e. the reflection center, change, the map needs to be recalculated. The resolution of the texture map is also important since it influences how accurately the reflection can be depicted. In addition, their accuracy depends on the distance between the reflected object and the reflector and will be better for more distant objects. According to McReynolds and Blythe [12] interreflections are possible by iteratively creating the environment maps for each reflector and then applying them for the next iteration.
More information and specific calculations for the mappings mentioned above can be found in the work of Mizutatni and Reindel [13] and in McReynolds and Blythe [12].
Building on these environment mapping methods, Yu et al. [20] developed a technique to improve on regular environment maps by using 4D *light fields* instead of 2D textures. Light fields are a collection of images on a 2D image plane. From those images, every possible viewing ray can be synthesized inside a given region, according to Yu et al. By surrounding the reflector with six such light fields, they can support dynamic reflections for moving reflectors inside the cube, including motion parallax.
Another extension to environment mapping can be found in Popescu et al. [16]. In addition to environment maps, they use two types of impostors to approximate the geometry of objects in the scene. The first type is the billboard. It approximates an object by mapping its image to a textured quadrilateral which can easily be intersected with reflected viewing rays. Optionally, they can also store surface normals per texel, to facilitate interreflections. The second type of imposter they use, is the depth map which is a billboard with an added depth channel. The depth maps improve reflections in cases where the object is close to the reflector or when the object and the reflector intersect. They also allow for motion parallax. Popescu et al. suggest that their method can be regarded as a middle ground between environment maps and ray tracing. The impostors

use more geometric information than the environment, but do not have as much geometric complexity as ray tracing.

2.2.2 Screen Space Techniques

A modern approach that has been developed in the last decade is called *Screen-Space Reflections* (SSR). The method was introduced as Real-Time Local Reflections by Sousa et al. [17]. This approach creates the reflections in a post-processing step. First, the scene is rendered into a buffer structure called *G-Buffer*. The G-Buffer is a collection of render targets that contains diffuse color information but also the geometrical information for each pixel of the rendered scene. It stores depth, surface normal and position values. After rendering to the G-Buffer, a ray is shot from the viewing position towards each surface point stored in the G-Buffer and then it is traced along its reflected ray using the stored surface normal. This ray is sampled at specific intervals. The sample points are mapped into the 2D screen space, where the sample point's depth is checked against the G-Buffers depth value. If the depth value in the G-Buffer is lower, this marks an intersection. An illustration of this process can be seen in Figure 5.

Sousa et al. [17] mention that, while it is a relatively fast technique, it can have problems due to the very limited information in screen space. McGuire et al. [11] give detailed information on the implementation of Screen-Space Reflections and improve the path sampling for more efficiency. A major drawback of this method is that it can only produce reflections of objects that are contained inside the current view. Tracing rays outside the image is not possible. This can lead to artefacts on the image boundary.

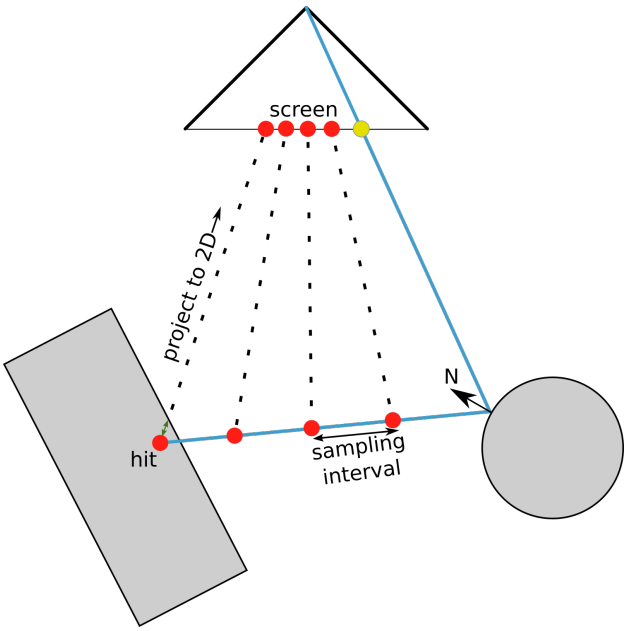

Figure 5: *A top-down overview of Screen-Space Reflection. A ray is shot through the current pixel in yellow. It is reflected using the surface normal N, that is contained in the current pixel of the G-Buffer. The reflected ray gets sampled and projected into Screen-Space. As soon as the sample depth is bigger than the depth in the G-Buffer, an intersection has been found. The color value of the intersection in the G-Buffer is then used for the final color in the current pixel.*

2.3 Real-Time Ray Tracing Techniques

An early implementation of ray tracing goes back to the work of Whitted [19]. He proposed a method for realistic rendering by following the viewing ray through the scene and recursively applying the intersection information to the current pixel. The number of how often the ray is reflected needs to be limited to keep the computation time low, but the higher the number the better the result. While this method works very well to create realistic images, it is computationally expensive. It heavily relies on visible surface algorithms to only test for intersections on an object if the ray crosses its bounding volume, instead of testing all objects in the scene. This technique alone is not sufficient for high-resolution images at interactive frame rates in complex scenes. There have been many improvements to this ray tracing algorithm, but only in recent years it was getting to a point where the results became real-time viable through advanced techniques and dedicated ray tracing hardware.

Bounding Volume Hierarchies (BVH) and *KD-trees* are essential for improving ray tracing speed, as they reduce the number of objects that need to be checked for intersection. An overview of these ray tracing data structures and architectures can be found in Deng et al. [5]. Recent advances have made it possible to construct a BVH in real-time as shown in Lauterbach et al. [9]. This allows for highly complex and dynamic scenes where the spatial data structure needs to change every frame. Denoising algorithms also greatly reduce the number of reflected rays needed per pixel to generate images without visible artefacts. State-of-the-art techniques for denoising ray traced images can be found in the papers by Bako et al. [1] and Marrs et al. [10]. Bako et al. use neural networks to denoise the images. Marrs et al. introduce an improved temporal antialiasing technique that uses adaptive ray tracing.
The most recent GPU architectures come with ray tracing cores that are capable of computing the above-mentioned algorithms in parallel directly on the GPU, allowing for notably faster image generation. Details on the most recent algorithms and architecture can be found in the Turing architecture white paper by NVIDIA [14].

To summarize, the biggest advantage of real-time ray tracing are the accurate reflections they produce. Interreflections are not only possible but inherently come with the algorithm. Despite the recent improvements real-time ray tracing is still dependent on dedicated hardware.

2.4 Hybrid Techniques

Each of the before mentioned techniques has its own shortcomings. Some of them can be avoided or alleviated by using more than one technique.
An early hybrid technique can be found in the work of Kilgard [8]. It combines planar reflections with stencil buffer clipping. This improves the clipping stage in certain cases. Interreflections, for example, can be done quite easily with the stencil buffer as it allows for marking individual pixels with the interreflection recursion depth.
Bastos and Stürzlinger [3] developed a hybrid approach that improves upon traditional environment mapping. They call it a hybrid between a geometry-based and an image-based solution. They warp the texture contained in the environment map into the space of the reflected viewpoint. In addition to the color information stored in the environment map, they also store the depth value of the texels. Their method preserves the depth of the reflected scene and corrects the perspective distortion that appears in classic environment mapping techniques. A detailed description for viewpoint warping can be found in their paper [3].
A more recent hybrid approach was proposed by Ganestam and Dogget [6]. They wanted to seamlessly trace paths in the scene, without using a full ray tracing approach. So, they developed a heuristic scene tracing approach. They divide the scene into different volumes. In a volume that is close to the camera, objects are placed inside a BVH (see Section 2.3), which is updated every frame. Outside this first volume, objects are rendered into a cube map structure of G-buffers. These buffers can be used for tracing the path in image space, reducing the complexity of the scene outside the innermost volume. Rays can be seamlessly traced between these two volumes. The combination of those two techniques is very efficient in avoiding the long computation times of ray tracing and the problems that come with the image-based buffer technique.
Walewski et al. [18] developed a method for hybrid rendering that determines which parts of the scene are to be rendered with secondary effects, like shadows and reflections, by calculating an importance value for them. They estimate the time it takes to render an object using ray tracing and weigh it against the importance value. Then they sort the scene into a graph, putting the more relevant objects at the top. When calculating the secondary effects, they start with the objects with the highest importance value and then follow the graph towards the most important objects that can still fit into the remaining available calculation time for the current frame. The importance value depends on multiple variables. Most of them are calculated every frame, like the size in the viewport, for example. Some are also determined by the user beforehand, for example, how important it is to select objects that were previously chosen for secondary effects. For a detailed description of how the importance value is calculated see the paper of Waliewski et al. [18].
The *PICA PICA* hybrid rendering pipeline is a hybrid rendering approach by Barré-Brisebois et al. [2] that combines traditional rasterization shaders with compute shaders and ray tracing shaders for the entire rendering pipeline. Their method does not specifically focus on reflections, but they are included as an integral part of their feature set. They state that reflections are one of the main features that benefit from ray tracing. Although they incorporated Screen-Space Reflections into their approach, they mostly use ray tracing for the final result to keep it simple. They also make use of denoising algorithms we previously mentioned in Section 2.3, that work on the final image to remove artefacts in areas where the number of traced rays was not high enough.

3 State-of-the-Art Rendering Engines

Currently, there are many real-time rendering engines publicly available. Most of them use state-of-the-art computer graphics techniques to portray realistic scenes and effects. Among those techniques, reflections are only a small subset of their capabilities, albeit a very important one. We will discuss two examples of freely available engines and compare their approaches and capabilities to give an insight into how they can produce real-time reflections. We chose these two because of their popularity and their extensive documentation.

3.1 Unreal Engine 4

The Unreal Engine 4 offers multiple different ways to produce real-time reflections. The first one uses planar reflections. This is Unreal Engine's geometric approach to render the scene a second time using a user-defined plane as a mirror. The engine handles clipping and reflective objects around the plane are taken care of automatically. This feature must be turned on deliberately in the engine's settings before it is available to the user, as it is potentially expensive to compute. Furthermore, they advise to only use a few of these planes if any at all, since it directly corresponds to the scene's complexity. To compensate for this the engine has multiple parameters to limit the number of reflected objects, for example, a maximum distance. More information can be found in the Unreal Engine Documentation on planar reflections [24].

The second method the Unreal Engine offers is Screen Space Reflections. This method is turned on by default. It generates little computational overhead as compared to other methods. There are only very few parameters to tweak the result, but the most notable one certainly is the quality setting that can be set between 0 and 100, with 50 as the default. The documentation [26] does not mention exactly how this parameter affects the algorithm.

The third option for reflections uses environment mapping. This method comes in multiple different forms. The Unreal Engine defines these as *Reflection Capture Actors* and *Scene Capture Actors* [23] that can be placed inside the scene. The former ones only map reflections inside a user-defined volume. This volume is either a cuboid or spherical. Their reflections are computed before run time and do not affect per frame computation time very much, since they are just environment maps which we already discussed in Section 2.2.1. The latter ones are fully dynamic cube maps. Their maps capture the entire scene and are recalculated on every frame, according to the documentation. This comes with a large computational cost. There is also the option for a 2-dimensional screen capture that works similarly but only maps to one texture instead of six cube map faces.

The final method for real-time reflections in Unreal Engine 4 is one that uses real-time ray tracing [25]. Its ray tracer is actually a hybrid between conventional ray tracing and raster effects, according to the documentation. A key ingredient for real-time viability is the denoising algorithm used by the engine. This allows for fewer samples during ray tracing.

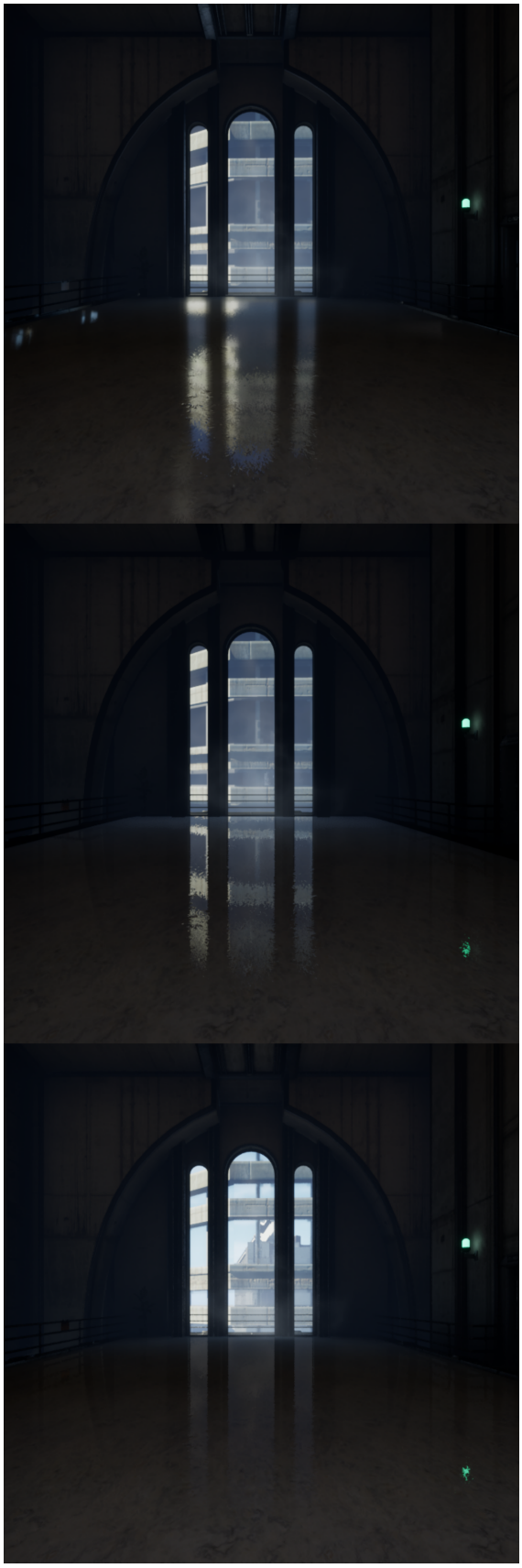

Figure 6: *A comparison of environment mapping (top), screen-space reflections (middle) and ray tracing (bottom) using Unreal Engine 4. The images are taken from the* BlueprintOffice *scene by Epic Games with the default rendering settings. The top image uses only* Reflection Capture Actors. *Notice how the reflection of the blue light source is not captured here. The reflections on the floor are blurry due to the limited environment map resolution. The windows of the building in the background are not encompassed by an environment map and therefore do not show reflections. In the middle image, only screen-space reflections are used. Thereby, the windows of the opposite building cannot show reflections because the outside walls of the room are not contained in the rendered image. The reflections on the floor are sharper because they use information from the rendered image directly. The bottom image uses ray tracing with a single bounce after the first intersection. The biggest difference in this image, compared to the other two, is that the windows of the building in the background show reflections of the exterior. The reflections on the floor are also sharper but much more subtle.*

Figure 6 shows a comparison of images that were created using different reflection techniques which are available in Unreal Engine 4.

3.2 Unity

Unity also supports multiple reflection techniques but their rendering engine is split into three separate rendering pipelines supporting different effects. When choosing a specific reflection technique, this has to be taken into account. See the Unity rendering pipeline documentation [22, 21] for a comparison between the rendering pipelines.

Similar to the Unreal Engine, Unity offers environment mapping in the form of cube maps. Here they are called *Reflection Probes*. They are placed inside the scene and can be used by any reflective object that comes close to the Reflection Probe. If there are multiple probes close to reflectors, the final reflection gets interpolated between their environment maps. According to the Unity documentation, this technique is available in every currently supported rendering pipeline, albeit with some minor differences.

Screen Space Reflections are available as a post-processing effect, but only in the High Definition Rendering Pipeline.

Real-time ray tracing is currently in preview and only available inside the High Definition Rendering Pipeline. Their approach is to completely replace other rasterized effects with ray tracing. This means that the ray traced reflections replace the screen space reflections. Additionally, ray tracing is not supported in combination with Reflection Probes.

4 Conclusion

Reflections in real-time scenes can be achieved in multiple ways. Geometry-based techniques can produce realistic results and are easy to calculate for planar reflectors, but curved surfaces are too complex to find a generalized solution. Image-based techniques can break the complexity of curved reflectors, since they work in image space rather than object space. Although not accurate, environment maps give a good approximate solution that can be calculated before run-time. Screen-Space Reflections work well for accurate reflections in real-time but are limited to the information of the camera view. Real-time ray tracing is getting more viable with dedicated hardware and improved algorithms to reduce tracing complexity. Hybrid approaches can compensate for the drawbacks of individual methods and can also produce fast and accurate results even though they can be more complex. Current state-of-the-art engines offer the user a variety of techniques to choose from to fit their individual needs.

References

[1] S. Bako, T. Vogels, B. Mcwilliams, M. Meyer, J. Novák, A. Harvill, P. Sen, T. Derose, F. Rousselle, Kernel-predicting Convolutional Networks for Denoising Monte Carlo Renderings, *ACM Transactions on Graphics (TOG)* **36**(4) (2017), 1–14.

[2] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak, J. Andersson, Hybrid Rendering for Real-Time Ray Tracing, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, Apress, 2019, 437–473.

[3] R. Bastos, W. Stürzlinger, Forward Mapped Planar Mirror Reflections, *University of North Carolina at Chapel Hill, Computer Science Technical Report TR98-206* (1998).

[4] J.F. Blinn, M.E. Newell, Texture and Reflection in Computer Generated Images, *Communications of the ACM* **19**(10) (1976), 542–547.

[5] Y. Deng, Y. Ni, Z. Li, S. Mu, W. Zhang, Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques, *ACM Computing Surveys (CSUR)* **50**(4) (2017), 1–41.

[6] P. Ganestam, M. Doggett, Real-time Multiply Recursive Reflections and Refractions Using Hybrid Rendering, *The Visual Computer* **31** (2015), 1395–1403.

[7] N. Greene, Environment Mapping and Other Applications of World Projections, *IEEE computer graphics and Applications* **6**(11) (1986), 21–29.

[8] M. Kilgard, Creating Reflections and Shadows Using Stencil Buffers, *At Game Developers Conference* **7** (1999).

[9] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH Construction on GPUs, *Computer Graphics Forum* **28**(2) (2009), 375–384.

[10] A. Marrs, J. Spjut, H. Gruen, R. Sathe, M. McGuire, Improving Temporal Antialiasing with Adaptive Ray Tracing, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (2019), 353.

[11] M. McGuire, M. Mara, Efficient GPU Screen-Space Ray Tracing, *Journal of Computer Graphics Techniques (JCGT)* **3** (2014), 73–85.

[12] T. McReynolds, D. Blythe, *Advanced Graphics Programming Using OpenGL*, Elsevier, 2005.

[13] Y. Mizutani, K. Reindel, *Environment Mapping Algorithms*, https://www.reindelsoftware.com/Documents/Mapping/Mapping.html, Accessed: 2021-8-4.

[14] NVIDIA, *NVIDIA Turing GPU Architecture*, White Paper, 2018.

[15] E. Ofek, A. Rappoport, Interactive Reflections on Curved Objects, *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), 333–342.

[16] V. Popescu, C. Mei, J. Dauble, E. Sacks, Reflected-scene Impostors for Realistic Reflections at Interactive Rates, *Computer Graphics Forum* **25**(3) (2006), 313–322.

[17] T. Sousa, N. Kasyan, N. Schulz, Secrets of CryENGINE 3 Graphics Technology, *SIGGRAPH, Advances in Real-Time Rendering in 3D Graphics and Games* (2011).

[18] P. Walewski, T. Gałaj, D. Szajerman, Heuristic Based Real-Time Hybrid Rendering with the Use of Rasterization and Ray Tracing Method, *Open Physics* **17**(1) (2019), 527–544.

[19] T. Whitted, An Improved Illumination Model for Shaded Display, *Proceedings of the 6th annual conference on Computer graphics and interactive techniques* **17**(1) (1979), 14.

[20] J. Yu, J. Yang, L. McMillan, Real-Time Reflection Mapping with Parallax, *Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), 133–138.

[21] *Unity High Definition Rendering Pipeline Documentation*, https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@12.0/manual/Feature-Comparison.html, Accessed: 2021-9-14.

[22] *Unity Universal Rendering Pipeline Documentation*, https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@12.0/manual/universalrp-builtin-feature-comparison.html, Accessed: 2021-9-14.

[23] *Reflections in the Unreal Engine Manual*, https://docs.unrealengine.com/4.26/en-US/Resources/Showcases/Reflections/, Accessed: 2021-5-27.

[24] *Planar Reflections in the Unreal Engine Manual*, https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LightingAndShadows/PlanarReflections/, Accessed: 2021-7-28.

[25] *Ray Tracing in the Unreal Engine Manual*, https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/, Accessed: 2021-9-22.

[26] *Screen-Space Reflections in the Unreal Engine Manual*, https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/ScreenSpaceReflection/, Accessed: 2021-9-22.

Christian Clemenz
e-mail: christian.clemenz@uni-ak.ac.at

University of Applied Arts Vienna
Oskar-Kokoschka-Platz 2, A-1010 Vienna, Austria

Leonard Weydemann
e-mail: leonard.weydemann@uni-ak.ac.at

University of Applied Arts Vienna
Oskar-Kokoschka-Platz 2, A-1010 Vienna, Austria