Adaptive Tessellation in Screen Space Curved Reflections

Attila Szabo[*]
Supervised by: Reinhold Preiner

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

Mirroring objects play an important role in the rendering of every-day scenes, as they aid in the recognition of materials, objects and the distance relations between them. Due to their complex nature, an accurate solution generally requires an expensive computation, which is mostly done using methods based on ray tracing. To reduce the workload, recent methods try to perform the computation in screen-space. However, in order to ensure accurate reflections, the geometry of the scene needs to be sufficiently tessellated to reduce the artifacts created by the linear interpolation of the GPU rasterizer. This creates a vast preprocessing effort and storage-overhead for the tessellated vertices.

In this paper, we present a method that performs this tessellation on the fly, reducing the error in the reflective image by inserting extra vertices where necessary. We prove the effectiveness of our approach in comparison to the state of the art and discuss limitations and ideas for possible future work.

Keywords: computer graphics, rendering, real-time rendering, deferred shading, tessellation, reflections

1 Introduction

In rendering, it is oftentimes the goal to render mirrors and reflective objects. Materials like metals and glass, make the scene feel believable to the viewer and contribute to the realism and beauty of images. Perfectly mirror-like surfaces can be often found in many man-made environments, such as in washroom appliances or cars.

In rendering systems, rendering specular reflections can be viewed as the process of finding *reflection points*, the places of reflection visible from the camera, and then reflecting the *reflected points* radiance at the reflection points toward the camera [10] (Figure 3(a)).

For rendering reflections, several methods and techniques have been developed. Usually, they either aim for maximizing *realism*, the physical accuracy, or *believability*, in which case the result will often only be ”correct enough” for it to look relatively accurate to the viewer, but gain faster performance, be easier for a designer to work with, look aesthetically more pleasing, or similar.

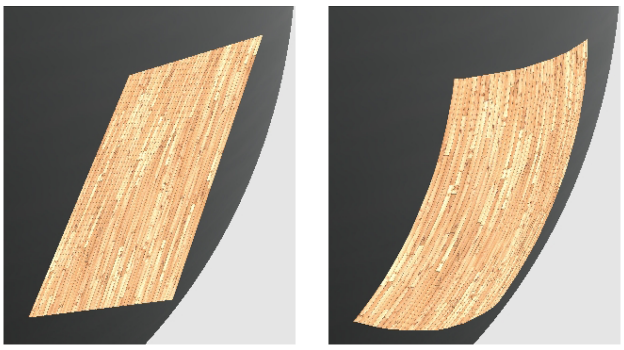


Figure 1: *Left:* Erroneous reflection of a square. Its sides are linear interpolations between the corners’ reflections. *Right:* Correct reflection after sufficient tessellation of the square.

The efficient rendering of planar mirrors has been examined extensively [7]. However, the case of **curved** surface reflectors requires special consideration since the reflections can become exceedingly complex. Light rays are traced up to the reflectors, the reflected rays computed and recursively traced until a non-specular surface is reached. In general scenes, the number of light rays and recursive computation steps can become very high. Therefore, effective storage of scene geometry and specialized processing of light rays are needed to guarantee robust performance.

CPU solutions usually create and maintain sophisticated scene data structures and optimized calculations to achieve good performance, while most GPU based techniques take advantage of the fast GPU rasterization capabilities [10]. The GPU processor is usually given access to the scene geometry by storing it in uniform parameters or in textures [10].

Both approximate [1] and accurate [11] methods have been developed to tackle this problem, trading performance for precision and vice versa. Screen-space methods for reflection rendering [4] are especially attractive since they are able to maintain both good accuracy and performance. Here, *Deferred Shading* [3] is used to relay the computation of a reflected image to a second rendering

[*]e0925269@student.tuwien.ac.at

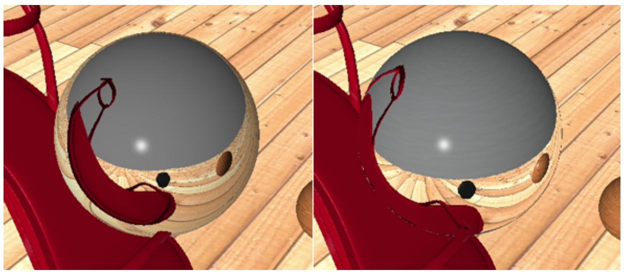


Figure 2: *Left:* Approximate reflection using Environment Mapping with errors. *Right:* Accurate reflection.

pass. First, the camera perspective of each mirror's surface is rendered into a series of textures. Then, the reflection of an object is rendered on top of it by mapping all its vertices onto their reflection points on a mirror's screen space projection (the *virtual geometry*) and exploiting the graphics hardware to triangulate the full reflected image.

This method is generally fast, but can lead to artifacts. In the virtual geometry reflected by the mirror, triangles can become curved patches, and triangle edges can become curves. However, the GPU is built to rasterize only non-deformed triangles with straight edges. A denser tessellation of one triangle reduces the size of triangles allowing edges to be closer to the curvature of the mirror. For the reflection to be free of artifacts, the geometry of the scene needs to be sufficiently tessellated, as shown in Figure 1. However, pre-tessellating a whole scene is generally unfeasible, since storing and processing a high number of vertices can quickly get computationally too expensive.

In this paper, we introduce an extension to this method to improve the visual quality of the resulting image. Our approach tessellates the geometry on the fly and only in the parts where it makes a visible difference. We control this mechanism by using a simple and flexible error metric. We show that our approach can efficiently produce accurate reflection images on curved mirrors without any pre-tessellation of the scene, while maintaining a good rendering performance.

2 Related Work

Accurate reflections are commonly computed using *Ray Tracing* [11]. In this approach, for every pixel, a number of viewing rays are cast into the scene and their interactions calculated. The main disadvantage of Ray Tracing is its high computational cost, since it generally requires a very high number of viewing rays and the interactions may be complex. *Online* rendering usually implies heavy performance constraints in order to retain interactivity. While improvements have been proposed to make the algorithm work effectively on graphics hardware [9], Ray Tracing is still not always suitable to provide interactive frame rates in many cases and mostly relies on building and maintaining spatial data structures [12]. This is especially problematic in *dynamic scenes*, in which these data structures have to be rebuilt or updated when objects move.

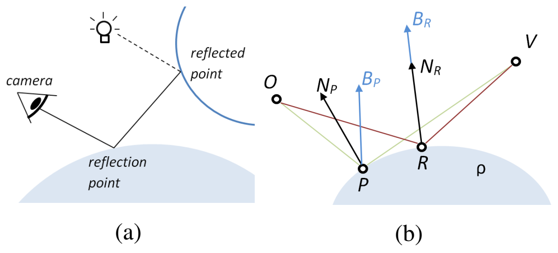


Figure 3: (a) Reflections are found by tracing light paths. (b) The relation between the viewpoint O, the world vertex V and a point P on the reflector surface ρ. The reflection point R is such that their bisector vector B_R and the surface normal N_R coincide.

Environment Mapping [2] allows an approximation of the reflection to be found very rapidly. In this approach, the environment around the reflector is rendered into a texture, such as a *cube map*. When shading a pixel belongig to a reflective surface, the reflected surface point in the environment is looked up in the cube map. However, this approach is not always physically accurate. Since environment mapping assumes that the environment is infinitely distant from the object, the reflection is approximately correct if the scene is sufficiently far away from the reflector surface [6, Chapter 7]. Figure 2 shows an example comparison between approximate and accurate reflections.

Reflections in planar mirrors are usually rendered by drawing the scene twice - once from the common viewpoint and once from the viewpoint reflected on the mirroring plane. The mirror image is then drawn on top of the mirror in the original image. *Non-Planar* mirrors however require a more sophisticated treatment, as their reflections cannot be modelled by linear projection as in the planar case above. The rendering technique proposed by Estalella et al. [4] addresses this circumstance. It follows the same idea of rendering virtual geometry inside a mirror, but extends it to curved mirrors. In the first rendering pass, the mirror's surface positions and normals are rendered into a series of textures. In the second pass, for each vertex in the scene, a pixel-by-pixel search across these textures is used to find the point that comes closest to its actual reflection on the mirror. To find this point, the following principle is used:

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 between O and V in P can be defined, as well as the curved reflector's *surface normal* N_P, see Figure 3(b). The *point of reflection* R on ρ is such that its bisector vector B_R and its surface normal N_R coincide. This point of reflection is unique across closed convex reflectors [5]. We use this principle later in Section 4.2 to determine adaptively the required degree of tessellation.

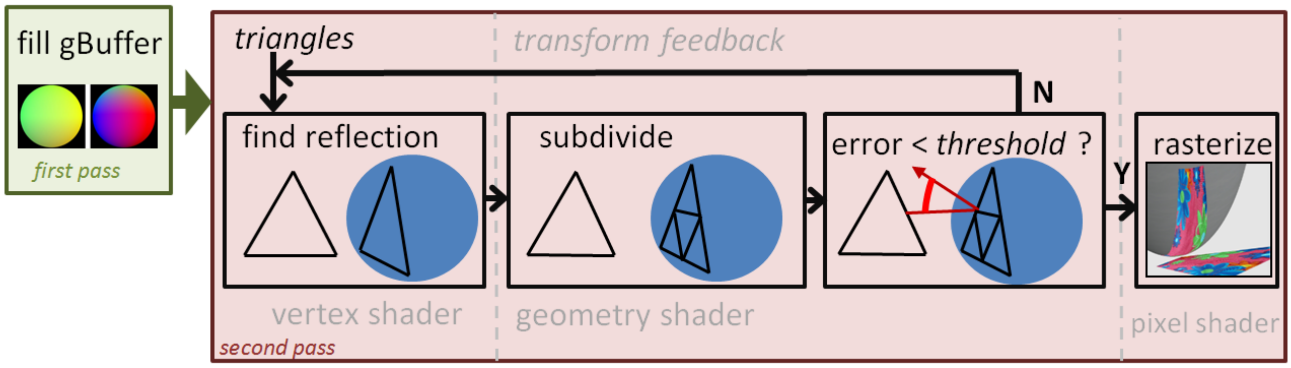


Figure 4: Overview of the rendering passes.

The result is a set of virtual ”mirrored” vertices, which are triangulated and rasterized on top of the original image. Although the mirror image is accurate for each of the vertices, their triangulation only represent a linear approximation to the correct non-linear curved mapping. Therefore, geometry must be tessellated fine enough for a sufficient piecewise linear approximation in screen space.

3 Overview

Reflections are rendered in a multi-pass approach. An overview is given in Figure 4. In the first pass, the mirror's surface positions and normals are rendered into the *gBuffer*. In the second pass, the gBuffer is used to calculate the reflection points of all vertices in the scene in the *vertex shader* stage (Section 4.1). The vertices are stored as vertex-triples forming triangles. The triangles are passed on to the *geometry shader* stage where based on a *reflection error* metric we check for each triangle whether whether it is sufficiently tesselated. If so, they are finalized for rasterizing. If not, the triangles are subdivided into four equal triangles and fed back to the vertex shader stage using *transform feedback*. The evaluation and subdivision of triangles is described in Section 4.2. The iterative subdivision of triangles continues until all triangles are finalized or an iteration limit has been reached. Finally, the triangles are rasterized to render the reflection.

4 Implementation

4.1 Screen Space Reflection

The system assumes the scene to consist of *triangle* primitives, which are marked to be *mirrors* or *non-mirrors*. Each scene object's vertices have a world-space position and a surface normal vector of the surface they belong to. The procedure of rendering accurate screen-space reflections for the mirroring scene is outlined by Algorithm 1.

Each reflector is rasterized from the camera's point of view, and its world-space position and surface normals stored in two 2D textures (*Position Map* and *Normal Map*).

```
foreach NonMirror n do
    Draw(n);
end
foreach Mirror m do
    gBuffer ← RenderGbuffer(m) ;
    foreach NonMirror n do
        DrawReflection(gBuffer,n);
    end
end
```

Algorithm 1: Functional outline of how a frame is rendered.

```
function FindReflectionPoint(gBuffer, vertex)
CurrentPixel ← MirrorCenter();
repeat
    PreviousPixel ← CurrentPixel;
    CurrentPixel ← BestNeighbour();
until PreviousPixel == CurrentPixel;
return GetPosition(gBuffer ,CurrentPixel);
```

```
function GetPixelError(gBuffer, Pixel)
S ← GetPosition(gBuffer,Pixel) ;
N ← GetNormal(gBuffer,Pixel) ;
po ← normalize( CameraWorldPosition - S);
pv ← normalize( VertexWorldPosition - S) ;
bisector ← normalize( po +pv) ;
return dot( bisector, N);
```

Algorithm 2: The method for calculating the reflection error of one pixel.

These two maps together are referred to as gBuffer [8, Chapter 9] in the following.

The next processing step is executed inside a vertex shader. The input of the vertex shader are individual vertices. The resulting vertices are passed on to the geometry shader. There triples of vertices are interpreted as triangles to be either drawn directly or subdivided, the resulting vertices being fed back into this stage.

Given a mirror's gBuffer and a vertex, Algorithm 2 shows how to find the vertex' reflection point on the mirror surface. The function starts its search at the the center of the reflector in screen space, and then iteratively steps towards the pixel position of the reflection point. At each currently considered pixel, its four directly neighbouring texels are examined and their reflection error calculated. This error is computed in the GetPixelError function.S and N are the position and the normal of the mirror surface point stored in this pixel, and po and pv are the direction vectors from the surface point S to the camera and the vertex position, respectively (see Figure 3(b)).

The deviation of the current pixel location to the sought reflection point (i.e., the *reflection error*) is measured by the dot product between the bisector vector between these two direction vectors and the reflector surface normal. This error is calculated for all four neighbours and the current pixel and then simply considers the neighbouring pixel with the lowest reflection error (highest dot product). This process is repeated until no neighbouring pixel with a lower reflection error current one can be found, at which point the final reflection point is found.

To ensure correct visibility when rendering a vertex, the z-buffer needs to be updated according to the reflected depth, i.e., the distance between the vertex and its reflection.

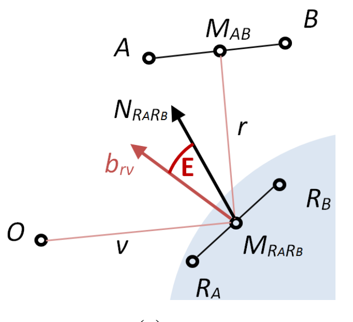
Some vertices do not have their reflection point on the visible surface of the reflector. For a reflector with closed uniformly convex geometry, such as a sphere, these are the vertices hidden behind the projection of the reflector in screen space. Taking such hidden reflection points into account for triangulation would result in incorrect triangles, and therefore have to be discarded. We identify such hidden vertices using the condition [4]:

$$pv \cdot N < 0$$

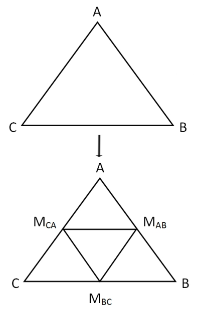
Vertices for which this condition is true are marked and their corresponding triangles are discarded. In addition, for reflectors which are not closed, such as reflectors that are partially obscured, the reflection point is found when the search terminates at the edge of the reflector projection [4].

4.2 Adaptive Tessellation

To address the problem of reflection artifacts for low-poly geometry (Figure 1), we perform an adaptive tessellation at render time.



(a)



(b)

Figure 5: (a) The reflection error E. (b) The subdivision rule used to tessellate a triangle.

This rendering stage is implemented in the geometry shader. The input are triples of vertices, forming *triangles*, resulting from the vertex shader stage (Section 4.1). The triangles are evaluated for subsequent subdivision. The resulting triangles are written to one of two vertex buffers, the *working buffer* and the *finished buffer*, using transform feedback. This rendering stage is outlined in Algorithm 3.

```
function adaptiveTess(A, B, C)
E1 ← calcError(A,B);
E2 ← calcError(B,C);
E3 ← calcError(A,C);
triError ← max(E1, E2, E3);
if triError < threshold then
    streamOut(finished);
else
    subdivide();
    streamOut(working);
end
```

Algorithm 3: Overview of adaptive tessellation function.

An input triangle is defined by its three vertices A, B and C, whose reflection points were calculated in the vertex shader stage. To decide whether a triangle is to be tessellated, the *reflection error* triError of the triangle is defined. The edge errors E are calculated for each of the three *edges*, formed by pairs of vertices, and triError is the maximum of the three E. For one edge, E is calculated in the function calcError using the following formula:

$$E = 1 - \frac{N_{R_AR_B} \cdot b_{vr} + 1}{2}$$

This relation is visualized in Figure 5(a). $N_{R_AR_B}$ 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

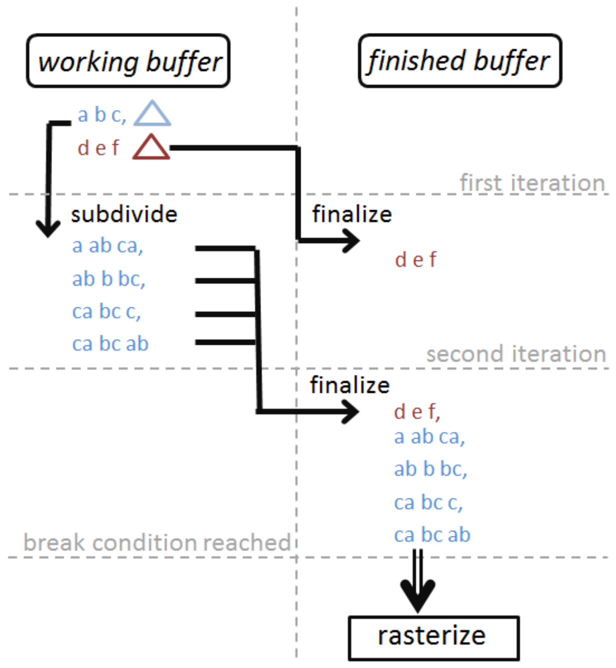


Figure 6: The tessellation loop iterates until the working buffer is empty or maximum iterations is reached.

normalized to lie inside $(0,1)$, where 0 means no error. E describes the difference between the linearly interpolated reflection of an edge and the one showing accurate curvature.

`triError` is compared against a user-set *threshold*. The threshold describes the highest acceptable deviation from the accurately curved reflection. If `triError` is less than the threshold, the triangle is deemed sufficiently tessellated and is *finalized*. Its three vertices are appended to the finished buffer using transform feedback (function `streamOut`).

If `triError` is greater than the threshold, the triangle is `subdivided` (function `subdivide`). The subdivision rule used is shown in Figure 5(b). The three halfway points along the edges M_{AB}, M_{BC} and M_{CA} are used as vertices with the three triangle vertices to form four equal triangles. The twelve vertices forming the four new triangles are written into the working buffer (function `streamOut`).

The rendering pass is repeated using the working buffer as input for the vertex shader stage. After their vertices being reflected, the triangles reach the geometry shader stage again to be finalized or further subdivided. This *tessellation loop* iteratively refines the tessellation of triangles until either no vertices are written into the working buffer, or a maximum number of iterations (the *tessellation level*) is reached. A visualization of the loop can be seen in Figure 6. If the iteration limit is reached, remaining vertices in the working buffer are copied into the finished buffer.

The tessellation level ensures that there is a hard limit to how often a triangle can be subdivided. The subdivision limit is usually not reached, unless the error threshold is set very low (close to 0), in which case subpixel accuracy is reached and further subdivisions can be limited. In addition, the limit avoids an infinite loop when the threshold is equal to 0.

After this rendering pass, the vertices in the finished buffer are rasterized. The result is a rendering of an accurate mirror image.

5 Results

5.1 Rendering Quality

Adaptive tessellation allows to render scenes as if they were fully tessellated. Figure 7 shows a scene being reflected in a mirroring ball. The scene contains both models with a very coarse and a very high original degree of geometry tessellation. Without adaptive tessellation (left), the result is visibly wrong. The tablecloth and candlestick are modelled using only a small number of quads. The corners of those quads are reflected correctly, but the linear interpolation between them does not suffice for a correct mirror image. The teapot is modelled with many more vertices and therefore produces a reflection image of much better quality. A full tessellation of the entire scene using four subdivision iterations is shown at the right image. For the teapot, this adds a lot of superfluous vertices, since the reflection does not improve. Using adaptive tessellation (middle) allows us to address both these problems. Coarse models are tessellated until quality of the rasterized reflection image is sufficient. On the other hand, geometry with already sufficient degree of detail, are not further subdivided, when drawing their reflection image.

Adaptive tessellation is very robust regarding different circumstances. Both simple reflections and complex surfaces are handled as accurately as needed. Our proposed error metric is derived from the screen space accuracy of the reflection and it relates directly to the errors visible in the rendered image. The error threshold represents a tradeoff parameter between quality and performance. It can be adjusted, even during runtime, to accommodate either faster performance or more accurate images. In fact, one could set the threshold to a sufficiently small value (subpixel size) to eliminate all visible artifacts.

As shown in Figure 7, our method can produce accurate reflections without the need for any pre-tessellation of the scene. It follows that content creators need not worry about specifically adjusting their models, and the technique can be implemented in a rendering system without big impact on established functionality.

5.2 Performance

The error threshold parameter allows for trading between accuracy and performance.

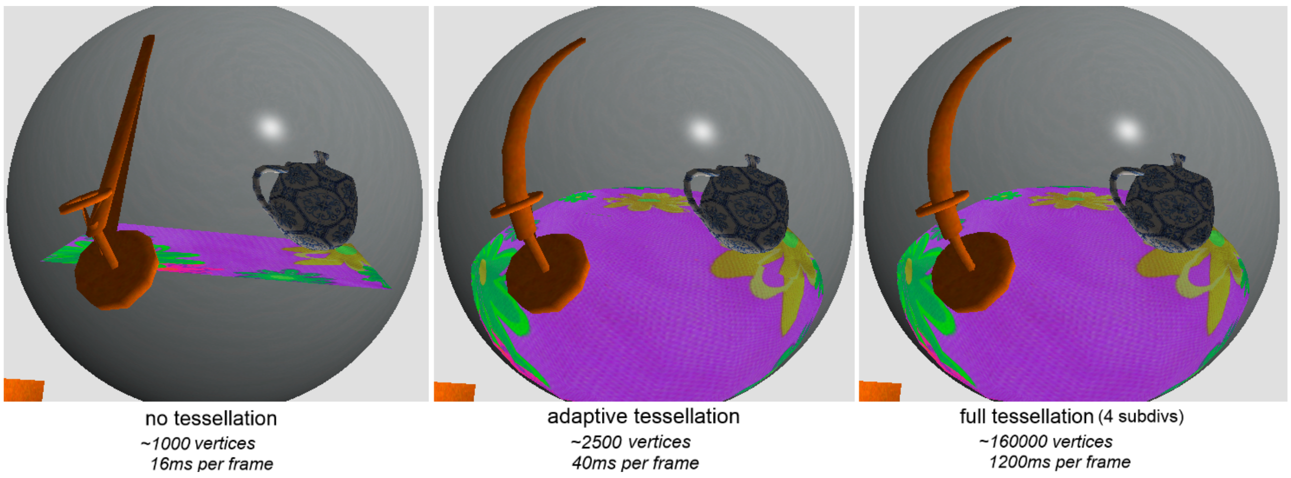
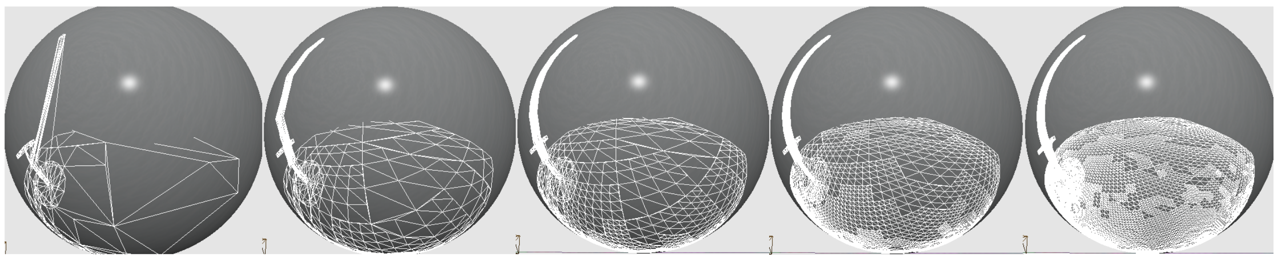


Figure 7: An example scene containing both coarse and fine meshes.



error threshold	0,1	0,01	0,001	0,0001	0,00001
vertex count	243	2376	5160	11058	36336
render time	25ms	40ms	75ms	120ms	400ms

Figure 8: Adaptive tessellation with different error threshold values.

Figure 8 shows the results of using different thresholds. The rendering has been performed on a PC with an *Intel Core2Duo* CPU and a *Nvidia GeForce GTX 260* graphics card. If the threshold is higher, larger errors are allowed and fewer subdivisions are performed. A too relaxed threshold results in a reduction of quality and introduces artifacts. We found that a value of 0.1 or larger is too high. A value between 0.1 and 0.01 generally creates perfectly acceptable results while maintaining goog performance. In particular, high curvature surface parts are subdivided often enough to provide accurate results. If the threshold is set even smaller, close to 0, the geometry is tessellated very finely. In this case, interactive performance can not be provided anymore. The value 0 itself causes *full tessellation* to be performed, in which case all triangle subdivision is repeated until the tessellation level is reached. Conversely, 1 stands for *no tessellation*.

Figure 9 shows a comparison of resulting vertex numbers between full, adaptive and no tessellation in the example scene (Figure 7). It can be seen that adaptive tessellation results in fewer vertices compared to full tessellation, requiring fewer expensive reflection point searches.

Furthermore, the performance of adaptive tessellation does not depend on any spatial data structures that cause a maintenance overhead. Therefore, dynamic scenes, in

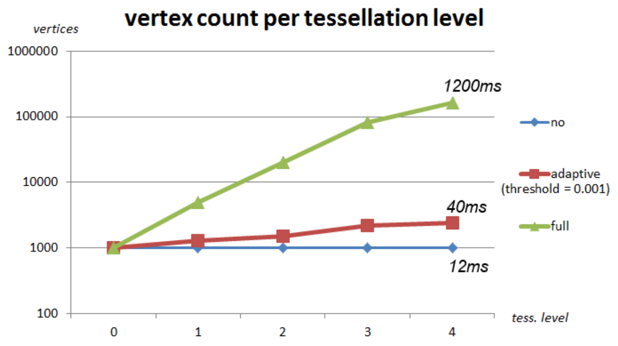


Figure 9: Number of vertices resulting from different tessellation levels in the scene from Figure 7.

which objects move or are otherwise animated, are handled without negative impact on the performance.

5.3 Limitations

As shown in Figure 10(a), our subdivision pattern can result in holes appearing in the reflection where triangles of different tessellations meet. The size of the hole relates to the difference in error of the two neighbouring triangles.

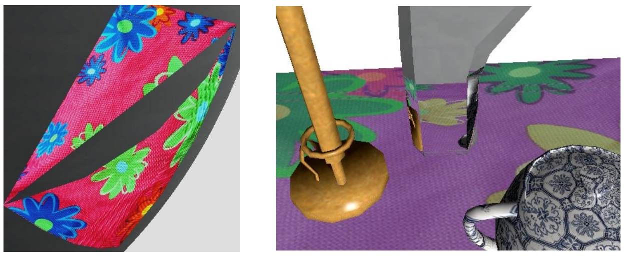

(a) (b)

Figure 10: (a) A hole between triangles with few and many subdivisions. (b) A simple polygonal mirror.

However, the error is only this severe if the camera is at a steep angle and a close distance to the reflector. In addition, with our error threshold the extent of the gaps is easily controllable. If the use of the threshold is restricted, one could instead imagine an extension in which the threshold is adjusted dynamically based on the probability of such holes appearing.

In addition, the base algorithm we use for finding reflection points assumes that vertices only have one reflection point [4]. As mentioned above, this holds for all convex mirror surfaces, and for concave surfaces of sufficiently large distances to the reflected geometry. Figure 10(b) shows an example for an arbitrary polyongal mirror.

5.4 Future Work

The problem of holes appearing between triangles could be solved by using different patterns of triangle subdivision. We use our subdivision rule because it is computationally simple and equally suited for any kind of reflector surface. However, it could be investigated to use irregular subdivision of triangles to achieve the same level of tessellation along shared edges, which would prevent holes from appearing between them.

Another improvement to the algorithm would be to extend it to arbitrarily shaped mirrors. This is a property of the underlying reflection point search algorithm. It could be solved by finding a method to split up the mirror into segments of uniform curvature and finding the reflection on each of them [4].

6 Conclusion

In this paper a method for rendering an accurate reflection on the surface of a curved reflector in real-time has been examined. It is a multi-pass approach in which first the image space reflection point of each vertex is found. The triangles formed by the vertices are tessellated adaptively according to an error metric, which is based on the difference in quality resulting from a subdivision iteration. Subdivision steps are skipped if they do not cause a noticeable effect in the final image, greatly reducing the number of vertices needing to be reflected. The subdivision is repeated until the triangles are sufficiently tessellated. Finally, the reflected geometry is rasterized by the graphics hardware. The method can provide interactive framerates for dynamic scenes. Discussed results show that the technique examined in this paper is a robust choice in real-time rendering and may well serve as an anchor point for future considerations extending its applicability.

References

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

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

[3] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 21–30. ACM, 1988.

[4] 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.

[5] Pau Estalella, Ignacio Martin, George Drettakis, Dani Tost, Olivier Devillers, Frederic Cazals, et al. Accurate interactive specular reflections on curved objects. In *Vision Modeling and Visualization (VMV 2005)*, 2005.

[6] Randima Fernando and Mark J Kilgard. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] Mark J Kilgard. Improving shadows and reflections via the stencil buffer. *Advanced OpenGL Game Development*, pages 204–253, 1999.

[8] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[9] 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.

[10] Lszl Szirmay-Kalos, Tams Umenhoffer, Gustavo Patow, Lszl Szcsi, and Mateu Sbert. Specular effects on the gpu: State of the art. *Computer Graphics Forum*, 28(6):1586–1617, 2009.

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

[12] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.