Light Culling Methods for Real-time Rendering Applications

Jared Watrous

October 24, 2024

Abstract

In real-time rendering applications that target realistic, high-fidelity output, light baking is the predominant solution in most modern pipelines. However, for scenes with many light sources whose properties (such as position or color) change over time, dynamic lighting at scale remains the primary method of rendering. As modern lighting models are computationally expensive, light culling is often used to reduce the number of per-pixel light computations. This paper surveys a variety of these methods and compares their performance benefits against their visual disadvantages to foster better intuition about the optimal use cases for each method.

Introduction

Real-time rendering applications for video games, film, concept visualization, and other forms of media often target visually realistic output with convincing materials and lighting. Many modern render pipelines rely heavily on baking, where visual effects such as lighting are precomputed offline and presented in a highly optimized format at runtime. These approaches can lead to very high-fidelity results, but they inherently create limitations in the types of scenes that can be rendered. Light baking, for instance, often relies on the assumption that light sources are static in position, direction, color, and even brightness, and varying any of these attributes can be difficult or infeasible under some architectures. As such, in applications that wish to change these properties frequently, it may be desirable to process the lighting dynamically, where the contribution of each light source is recomputed every frame.

Light Culling. Compared to baked lighting, dynamic lighting is expensive to render. In a trivial pipeline, render time scales linearly with the number of lights, as each rasterized pixel sums the contribution of every light source in the scene. Light culling is the process of pruning these light computations by deciding which ones are necessary for high-fidelity output. Most light culling algorithms rely on the concept a **light volume**, the region of space in which a given light source is assumed to have non-negligible contribution. This paper focuses primarily on simple point light sources, where the light volume can be defined as a simple bounding sphere. To determine its radius, we call that light attenuation follows the inverse square law:

$$\text{Attenuation} = \frac{1}{\text{distance}^2}$$

Figure 1: *Two light volumes visualized as polygonal spheres in the render engine used for this project, using an attenuation threshold of 0.3. In practice, a smaller attenuation threshold would be more typical for light culling; for evaluation, this paper uses a threshold of 0.02. Image contrast is increased for clarity.*

In this paper, we assume point lights follow exactly this inverse-quadratic attenuation. Following the above formula, we can easily derive a lower bound on the radius of the light's influence given some attenuation threshold:

$$\text{Radius} = \sqrt{\frac{|\text{Light}|}{\text{Threshold}}}$$

The light culling methods surveyed in this paper treat the above spherical light volume as a black-box definition and does not account for any other information about the light.

Pipeline Architectures

In general, most render pipelines can be placed into one of two categories.

Forward Rendering. Forward renderers directly rasterize the geometry data to an output image, often

either to the screen itself or to some intermediate framebuffer for post-processing. In forward rendering, typically only one pass of fragment shaders are executed, in which materials are evaluated and all light computations take place, including light culling. Optionally, some pipelines may choose to perform a depth pre-pass, where all the geometry is drawn once without a fragment shader (or with a very simple one) to fill the depth buffer, and then a second time to fully compute the output image. The benefit of a depth pre-pass is the reduction of overdraw, when a pixel is (expensively) drawn to more than once.

Deferred Rendering. The other category of pipelines, deferred renderers [1], add a layer of indirection by first rendering the scene's geometry to a "G-buffer," a temporary intermediate buffer that stores positions, normals, and material attributes for each pixel. In a second pass, these geometry components can then be read back and combined to produce an output image. Excepting their limitations (i.e. transparent objects), deferred renderers effectively eliminate overdraw by only computing lighting for pixels that are guaranteed to be visible in the output image. By decoupling material evaluation from lighting, it also permits lighting computations to be performed on arbitrary regions of the image.

Many light culling algorithms can be integrated into both forward and deferred render pipelines.

Principles and Constraints

In surveying light culling methods, this paper strives to compare them with respect to a set of practical principles and constraints that (ideally) should be relevant to most use cases for the algorithms.

Net-Nonnegative Performance. Ideally, a light culling algorithm should lead to reduced frametime on typical scenes where lighting is of particular concern. In contrast, a light culling method that is typically slower than simply computing every light provides no value to the application. Applications with few or static lights may not opt for large-scale dynamic lighting, so we are particularly interested in the performance as measured on scenes with lots of dynamic light sources.

Scene-Agnostic. Ideal methods should not need adjustment when moving from scene to scene, and should not depend on any of a scene's contents other than light volumes and the geometry to be illuminated.

Transparent to the Artist. The artist should not have to consider light culling when designing a scene for the render engine.

Unbounded Number of Lights. Following from transparency, an ideal algorithm imposes no hard limits on the number of light sources in a scene. Hardware limitations such as memory usage are exempted.

Minimal Visual Artifacts. Light culling inherently imposes approximations, as it assumes a light volume is representative of a light's influence. As such, visual artifacts are plausible with many algorithms. An ideal algorithm minimizes these artifacts as much as possible to maintain the fidelity of the output image.

The purpose of this paper is to compare various light culling methods with respect to these principles and constraints.

Methods

The following methods were considered in this survey. Where possible, both forward and deferred renderers were evaluated.

No Light Culling. In this baseline method, no light culling was used. During the lighting phase in fragment shaders, the shader iterated over every light source in the scene and added its contribution to the output pixel.

Bounding Sphere. Before computing a light's contribution, the fragment shader first checks whether the current pixel's position is inside the bounding sphere of the light volume. If so, the shader computes the light as normal; if not, the shader skips the light source. In this method, the fragment shader still iterates over every light source, but only computes the subset of lights whose volumes contain the current pixel.

Raster Sphere. Unique to deferred renderers, this method rasterizes each light volume as a world-space sphere [2]. In the sphere's fragment shader, the G-buffer is read and used to compute the contribution of the light associated with the sphere. Additive blending is then used to combine the results of all rendered spheres from every light source. To prevent the light from disappearing if the camera enters its light volume, front-face culling may be used to only render the interior of the sphere. As an additional optimization, "greater-than-or-equal-to" depth testing can be enabled, which only draws pixels where

the light volume's far bound is behind some existing geometry (i.e., there exists some geometry inside the light volume at that pixel.) The Raster Sphere method benefits from only computing the light source for exactly the pixels contained in its light volume, and avoids excess loop iterations in other pixels. The tradeoff is that extra geometry is rendered, and especially if the rendering backend is serial, this may become a bottleneck when there are many lights.

Clustered Rendering. This method is an extension of tiled rendering. In tiled rendering [3], the output image is divided into screen-space voxels called tiles. As a per-frame preprocessing step, interesections between all light volumes and each tile are computed, and a per-tile list of intersecting lights is generated. This is often done in a compute shader on the GPU. During rendering, the fragment shader determines which tile the current pixel is in, then queries that tile for the list of lights that intersect it. Clustered rendering [4] extends this approach by additionally subdividing the camera frustum in the depth direction, yielding 3-dimensional clusters rather than 2-dimension tiles, and the rest of the algorithm is identical. This approach permits directly looping over lights in the fragment shader without redundant iterations, and assigns lights to these lists in an automated, parallelizable manner.

One technical limitation of the clustered rendering implementation evaluated in this paper is that, due to the nature of OpenGL's buffer allocation, there is a hard-coded limit on the number of light sources that can influence each individual cluster. This conflicts with the principle that an ideal algorithm should support an unbounded number of lights, and thus is a significant disclaimer for this algorithm. Nevertheless, because it still imposes no scene-level limitations, and the likelihood of hitting the per-cluster maximum is relatively low in natural scenes (and such an event can be handled gracefully by simply ignoring any lights past the maximum), this paper still considers the method to be viable option for many applications.

Implementation

All four methods discussed above were implemented into an existing lightweight rendering framework written in C++17 with an OpenGL backend.[1] No light culling, Bounding Sphere, and Clustered Rendering were integrated into an original forward render pipeline, and all of these plus Raster Sphere were integrated into a deferred render pipeline. The forward render pipelines uses a depth pre-pass. The Raster Sphere method uses "greater-than-or-equal-to" depth testing. Clustered rendering uses subdivisions of 48 by 27 by 24 (where each screen-space tile is 40 by 40 pixels at 1080p resolution) and has a maximum of 128 lights per cluster. The compute shader for clustered rendering uses a bounding box approximation to test frustum-sphere intersections as described in [5]. In all cases, the attenuation threshold used to compute light volumes is 0.02.

Lighting Model. Lighting calculations were performed using a Physically-Based Rendering (PBR) model with a Cook-Torrence BRDF [6].

Non-priorities. The render engine used for this project **does not** support any visual effects beyond those described here, with the exception of basic dynamic range and gamma correction. Specifically, it does not support shadows, frustum culling, or any screen-space effects such as reflections or ambient occlusion.

Clay Renderer. As a control, a "clay" render pipeline was also implemented. This pipeline, separate from the primary forward or deferred pipelines, is a bare-bones forward renderer that rasterizes the geometry to the screen without performing any complex operations in the fragment shader. Rather, it simply draws a solid-color scene with no textures and only a constant-time Phong lighting model (with a false view-space directional light.) The purpose of this control is to gauge the performance of the boilerplate render engine without confounding it with the light computations.

Evaluation

To evaluate the performance of the light culling methods, I prepared a test scene using PBR assets released under the CC0 license and a fixed camera trajectory consisting of 1000 frames. I also fixed 7 "spawn points," bounding cylinders in which lights may spawn at program execution.

During evaluation, the program is parameterized with a set number of lights. Each light is spawned by selecting a "spawn point" uniformly at random, choosing a location inside the bounding cylinder uniformly at random, and choosing a color with uniformly random hue and saturation. The program then renders each of the 1000 frames of the camera trajectory in sequence and displays it in a windowed

[1] This lightweight framework is an old unfinished repo of mine that I repurposed for this project. It already supported scene graphs, asset and texture loading, and simple camera controls. For this project, I implemented complete forward and deferred render pipelines, as well as all of the described light culling methods.

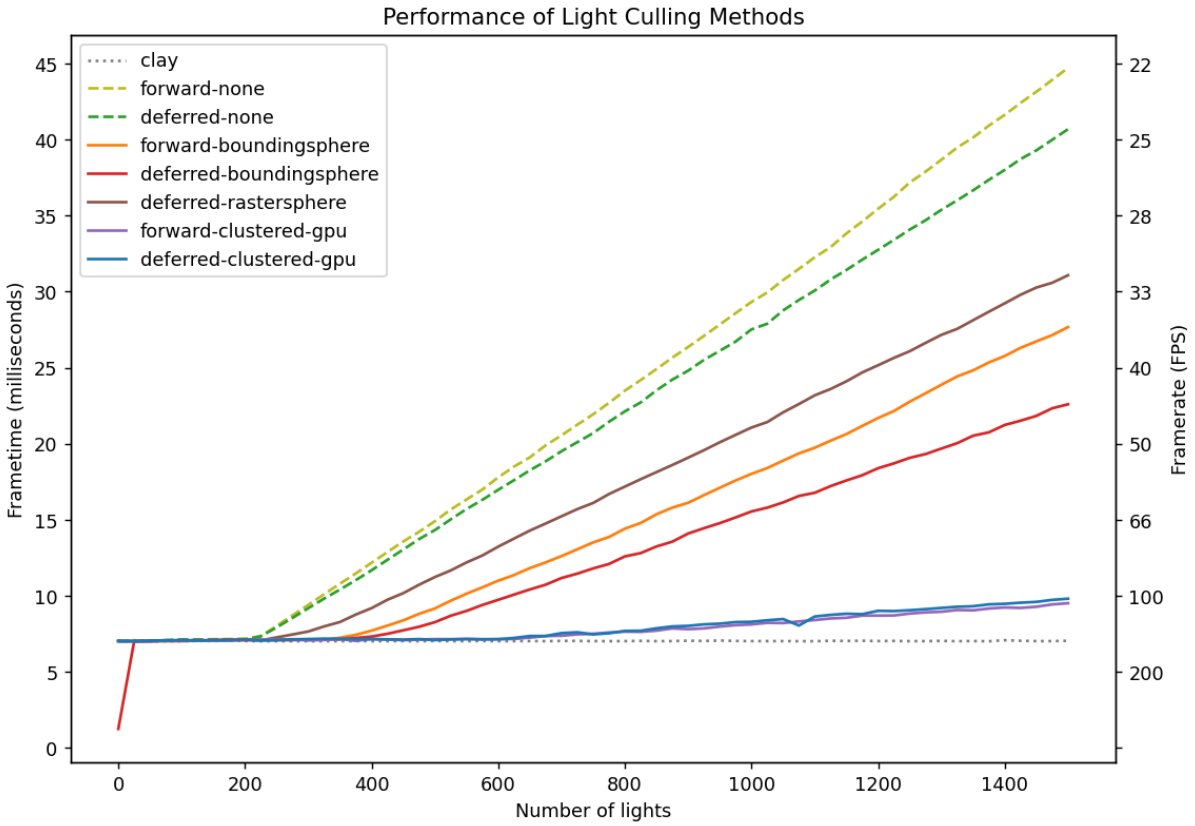

Figure 2: *Average recorded frametimes of each light culling method with different numbers of lights.*

application. Frametimes are recorded, including the execution time of all boilerplate code, such that the sum of all frametimes equals the wall-clock time of the program's execution. VSync is disabled during evaluation. All benchmarks were recorded on the same computer with an AMD Ryzen 5900X 12-core processor and an NVIDIA RTX 3080 GPU.

Figure 2 presents the average recorded benchmarks of each method with the number of lights ranging from 0 to 1500. As one would likely intuitively expect, the runtime of each algorithm increases linearly with the number of lights. All methods are bounded below by the clay renderer, and the two methods (forward and deferred) without light culling are the slowest. The Bounding Sphere methods offer a very significant speed-up, cutting the overhead of light computation nearly in half. Raster Sphere, in rendering more geometry, delivers slightly less of a speed-up. Notably, with both Bounding Sphere and without light culling, the deferred renderer reached higher performance (lower frametime) than the forward renderer. In both renderers, clustered rendering greatly outperformed all of the other methods, reaching up to 1500 lights at a comfortable 10 ms per frame (100 frames per second.)

Failure Modes: Reflective Surfaces

Recall that all of the light culling methods explored in this paper rely on the assumption that any pixel outside a light volume is only negligibly influenced by that light source. In a PBR lighting model, this assumption may hold well for dielectric materials with sufficiently high material roughness. But for metallic materials or dielectric materials with low roughness, the assumption can be easily broken. A trivial example is a perfect mirror, where pixels on the mirror can be illuminated by light sources arbitrarily far away. In practice, this can lead to significant visual artifacts when light culling methods are used. Figure 3 presents some examples of these artifacts found with the light culling methods evaluated here. Reflective surfaces are further discussed in the Further Work section.

Conformance to Principles and Constraints

Each of the light culling methods explored here are naturally scene-agnostic and easily achieve net-nonnegative performance. With the exception of clustered rendering, they are also completely transparent to the artist and support an unbounded number of lights. Although the clustered rendering implemen-

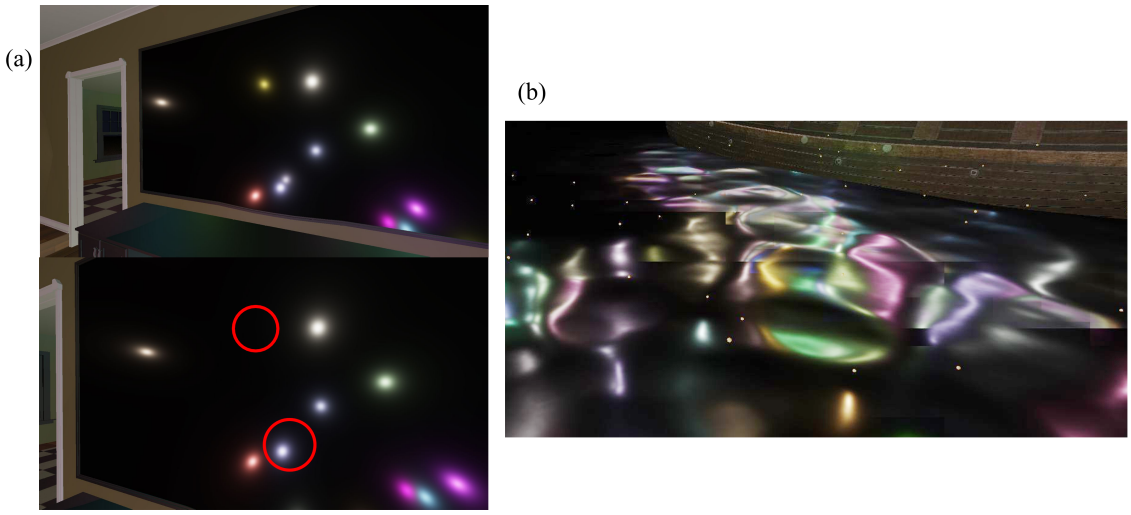

Figure 3: *Artifacts produced by light culling.* ***(a)*** *shows some light sources disappearing in the reflection as the camera moves around the scene. This phenomena can be observed with both Bounding Sphere and Raster Sphere culling (here Raster Sphere is visualized, and the red circles were added to highlight the differences.)* ***(b)*** *shows grid-shaped tiling artifacts on a reflective surface, visible in Clustered Rendering (image contrast increased for clarity.)*

tation has an implicit cap on the number of lights per cluster, as described in the Methods section, hitting this maximum is likely to be rare. An artist may need to be conscious of grouping hundreds of light sources close together if they wish to completely avoid artifacts, but this constraint is somewhat unnatural and a typical scene is unlikely be problematic under this limitation. As for visual artifacts, none of the methods are perfect, especially with reflective surfaces. However, the Bounding sphere method in particular, if it produces artifacts at all, is likely to produce smooth, circular banding effects, which may be less noticeable or distracting than tiling artifacts possible with clustered rendering.

Future Work

The light culling methods explored in this paper are fairly well-known methods to reduce the overhead of dynamic lighting. Clustered rendering in particular is used by most game engines in practice. Future comparative work could include expanding to more light culling methods, such as stochastic light culling. Research in the area, however, seems to have shifted towards the integration of dynamic lights into baked systems, such as with Activision's UberBake [7].

Mitigating Reflective Surfaces. The issue of reflective surfaces stems from the fact that the light volume assumption is inherently flawed, as it assumes properties about the materials of the illuminated geometry. If some lower bound on the reflectivity of all materials was known, then the light volume assumption could be fine-tuned to a particular scene by adjusting the threshold accordingly. But even this approach would have its issues; not only are the material properties of a scene generally unknown and (in the spirit of scene-agnosticism) hard to efficiently evaluate, if all light volumes were determined with respect to a scene-level lower bound, then a single highly reflective material, however small, could inflate all light volumes to an arbitrarily large size, killing the benefit of light culling. In effect, the introduction of material properties into the equation transforms the light volume definition from a light-dependent problem to a light-and-material-dependent problem.

Recall that in clustered rendering, we consider a single cluster (a slice of a frustum in 3-dimensional space) to be representative of all screen-space pixels in that cluster. In other words, light volumes that intersect the cluster are accumulated into a single list such that every pixel in the cluster computes the same set of lights. To transform this setup into a material-dependent solution, one option is to similarly accumulate material properties within a cluster. For example, consider a deferred render pipeline. After rasterizing geometry to the G-buffer, but before rendering to the screen, a compute shader is run to compute the light lists for each cluster. In this compute shader, all pixels inside the current cluster are read from the G-buffer (that is, discarding pixels outside the screen-space tile and outside the depth range of the cluster.) The material properties for these pixels, read from the G-buffer, are accumulated into a summary of values (e.g., a lower bound on reflectivity), and this summary is used to adjust the light volumes of all lights when testing intersection with the cluster.

The algorithm described above is only speculative,

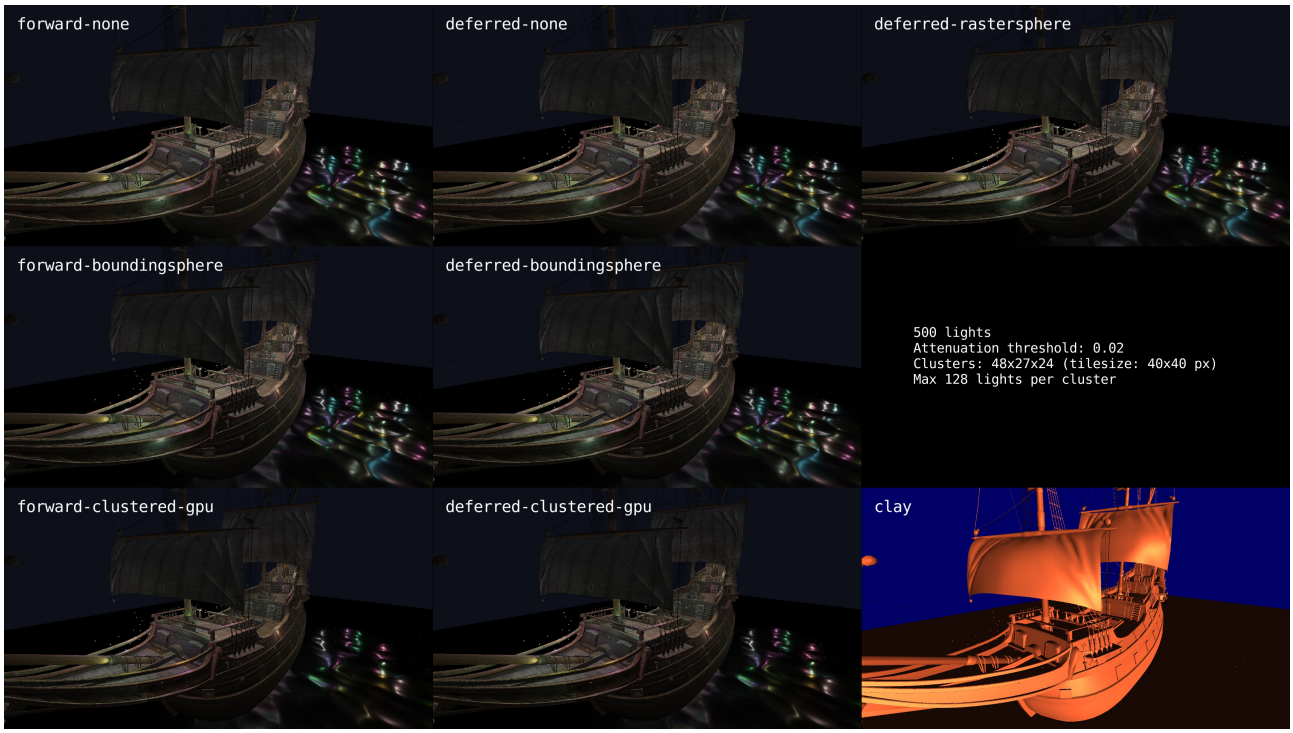

Figure 4: *Sample images from each light culling method presented in this paper, including the "Clay" control renderer.*

but perhaps worthy of exploring in future work. It still fails to address changes in lighting with respect to viewing angle, and only benefits deferred renderers and scenes with dynamic lighting and reflective materials.

Supplementary

Figure 4 shows sample screenshots from each light culling method described in this paper. A full video sample is available at:
`https://drive.google.com/file/d/1g5uZtsiuNaM2lRFc9fFTxfLeDlTR3Edu/view?usp=sharing`

The source code for this project is available on GitHub:
`https://github.com/thetruejard/cs348k_project`

References

[1] Jonathan Thaler. *Deferred Rendering*. Jan. 2011.

[2] Joey de Vries. "Deferred Shading". In: (2015). URL: https://learnopengl.com/Advanced-Lighting/Deferred-Shading.

[3] Takahiro Harada, Jay McKee, and Jason C. Yang. "Forward+: Bringing Deferred Lighting to the Next Level". In: *Eurographics 2012 - Short Papers*. Ed. by Carlos Andujar and Enrico Puppo. The Eurographics Association, 2012. DOI: 10.2312/conf/EG2012/short/005-008.

[4] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered deferred and forward shading". In: *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings* (Jan. 2012), pp. 87–96. DOI: 10.2312/EGGH/HPG12/087-096.

[5] Angel Ortiz. *A Primer On Efficient Rendering Algorithms Clustered Shading*. Dec. 2018. URL: https://www.aortiz.me/2018/12/21/CG.html.

[6] R. L. Cook and K. E. Torrance. "A Reflectance Model for Computer Graphics". In: *ACM Trans. Graph.* 1.1 (Jan. 1982), pp. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. URL: https://doi.org/10.1145/357290.357293.

[7] Dario Seyb et al. "The design and evolution of the UberBake light baking system". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 39.4 (July 2020). DOI: 10/gg8xc9.