Intersection Primitives

May 29, 2023

Andrea Bernini

1. Introduction

The project focused on implementing an improvement for the intersections of some roughly calculated primitives within the `yocto-gl` library. To be precise, it was requested to render **Points as Spheres**, **Lines as Rounded Cone** and **Quads as Bilinear Patches**, using as references the work done by Quilez for spheres and cones, and the work of Reshetov for bilinear patches.

2. Related Works

Cool Patches Intersections between a ray and a nonplanar bilinear patch using simple geometrical constructs. The new algorithm improves the state of the art performance by over 6× and is faster than approximating a patch with two triangles (Reshetov, 2019).

Ray Tracing Generalized Tube Primitives A general high-performance technique for ray tracing generalized tube primitives. Our technique efficiently supports tube primitives with fixed and varying radii, general acyclic graph structures with bifurcations, and correct transparency with interior surface removal (Han et al., 2019).

3. Method

The project outline presented two possible approaches to implement the improvement. The first method is to modify the `eval_position()` and `eval_normal()` functions (in `yocto_scene.cpp`) to compute the intersection position and normal vector for the new primitives. The second method, instead, proposes to directly return the position and the normal in the calculation of the intersection.

Initially, the second option was chosen due to its simplicity. Once created the three new intersection functions for spheres, cones, and patches (in `yocto_geometry.h`), it was enough to modify the return type of the intersections primitive function (`prim_intersection` in `yocto_geometry.h`, and consequently also the `scene_intersection` and `shape_intersection` structures), so that **position** and **surface normal** can be returned. Thus, in the path tracing functions in `yocto_trace.cpp`, position and normal could be accessed directly without calling `eval_shading_position()` and `eval_shading_normal()`.

However, this approach involved a problem in managing the calculation of the surface normal for Bilinear Patches, because to calculate it correctly, it was also necessary to take into account the **material type** (as can be seen in `eval_shading _normal()` in `yocto_scene.cpp`). This would have meant adding more parameters to the intersection calculation function with the primitive Bilinear Patch, making the code less readable and more logically complicated.

For this reason, I initially decided to implement the calculation of position and normal, in `eval_position()` and `eval_normal()` for bilinear patches, and leave it within the intersection functions for Spheres and Rounded Cone, handling the two different approaches within the path tracing functions (in `yocto_trace.cpp`). After getting good results in testing, I still decided to implement position and normal handling of the spheres and cones in `eval_position()` and `eval_normal()`, to make the code more uniform.

Bilinear Patch As for the Bilinear Patches, the code provided by Reshetov's work was used, adapting it to the syntax and functions of `yocto`, by implementing the intersection function `intersect_patch()` in `yocto_geometry.h`, and the calculation of normal and position in `yocto_scene.cpp`.

Sphere To calculate the intersection (`intersect_sphere()` in `yocto_geometry.h`), I reused the code provided by Quilez, while to calculate the UV texture coordinates, I reused the vanilla `yocto` approach, i.e., using the **spherical polar coordinates** (Dodgson & Blackwell) to convert the surface normal to U and V. In this way, in `eval_position()` and `eval_normal()`, I can do the reverse procedure (from

Email: Andrea Bernini <bernini.2021867@studenti.uniroma1.it>.

Fundamental of Computer Graphics 2022, Sapienza University of Rome, 2nd semester a.y. 2021/2022.

polar spherical coordinates to Cartesian coordinates) to obtain the normal and therefore also the position using the inverse formula for calculating the normal of a sphere ($Position = Normal + Center$, from $Normal = Position - Center$).

Rounded Cone To calculate the intersection between the ray and a Rounded Cone, as with the spheres I used the code provided by Quilez, implementing it in `intersect_cone()` (in `yocto_geometry.h`). While to calculate the texture coordinates I used the **cylindrical polar coordinates** (Dodgson & Blackwell), in such a way as to be able to make the conversion into Cartesian coordinates in `eval_position()` and `eval_normal()`, (which with the vanilla `yocto` method, used for lines would not have been possible). Using this approach I can obtain the position, and based on this, calculate the surface normal, managing three different cases.

In the first two cases, the intersection with the ends of the cone represented by spheres is handled, calculating the normal vector accordingly ($Normal = Position - Center$). In the third case, using calculations taken from the Han et al.'s work, it was possible to determine the apex of the virtual cone if the two spheres had different radii, and then calculate the normal of the cone on the basis of this information, otherwise if the two spheres have the same radius the normal of a cylinder will be calculated.

4. Experimental Results

To choose which primitives to use, I introduced checkboxes (Figure 1) that can be selected in the graphical interface in order to facilitate the execution of comparison tests.

From Figure 2 we can see that there is a marked improvement in the use of Spheres instead of dots, in fact, the two grids of dots are better visible (the second and third figures starting from the left).

Even for the lines, treated as Rounded Cone, we can say that we have obtained a good improvement in that the shadows result more homogeneous than the light sources (Figure 7), or their color is more evident (Figure 2).

Finally, as regards the quads treated as bilinear patches, no major changes are observed, perhaps due to the lack of complexity of the figures (Figure 9), and as described in (Reshetov, 2019), the use of patches speeds up rendering, however, the tests in yocto revealed a difference (on average over 10 trials) of about 20 seconds (03:29.736 minutes on average for yocto vanilla and 03:09.713 for yocto enhanced) very far from the 6x speed increase described in the paper.

The code, with all other test images, is available at FoCG-project.

5. Conclusion

The use of these new primitives certainly brings advantages to the visualization layer, however, there are also disadvantages. For example, the Ray-Point intersection is less expensive than the Ray-Sphere intersection, as it has fewer multiplication and square root operations (but this is not the case with the function written by Quilez). However, the main drawback has to do with the numerical accuracy, in fact Ray-Point Intersection is numerically more accurate, as Ray-Sphere Intersection includes square roots for its development, so if floating point errors are made these they will be greater.

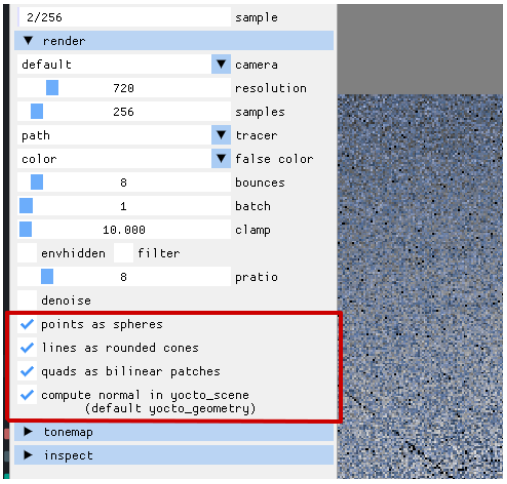


Figure 1. Switch to select intersection mode (red box).

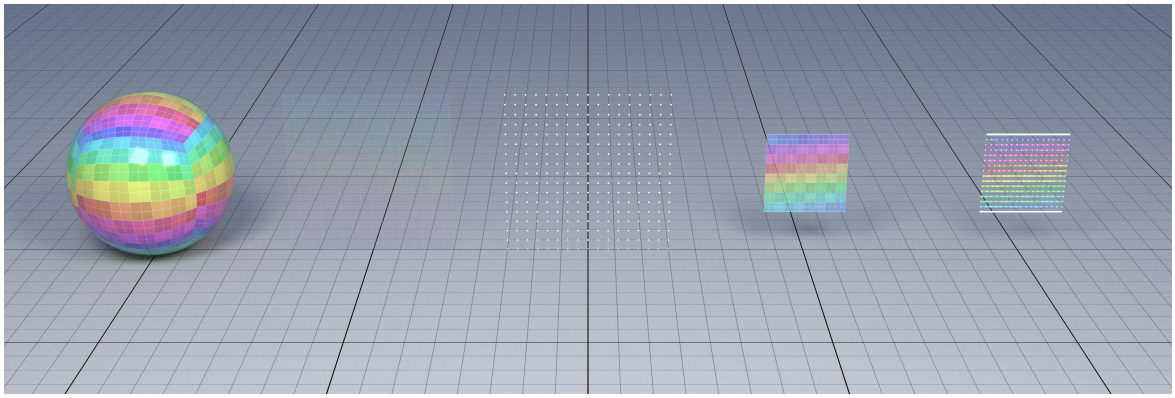
References

Dodgson, N. and Blackwell, A. Ray tracing primitives. https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html.

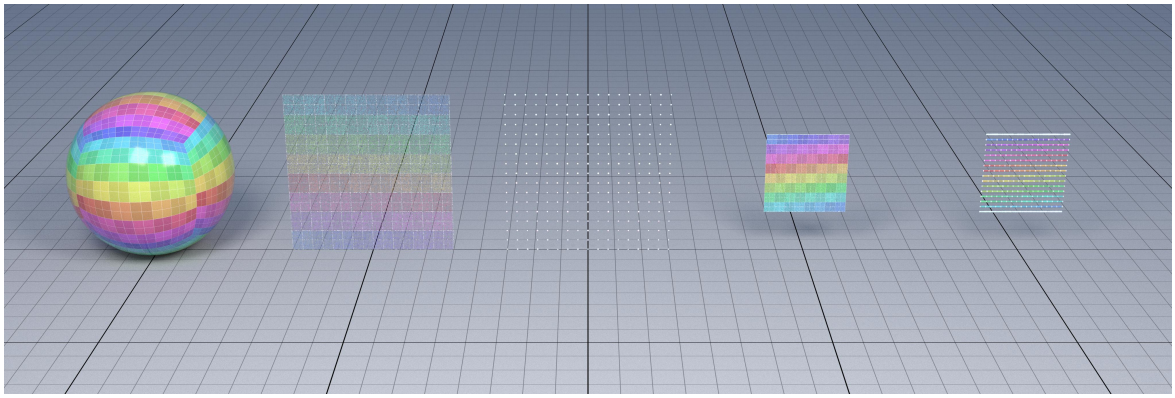
Han, M., Wald, I., Usher, W., Wu, Q., Wang, F., Pascucci, V., Hansen, C., and Johnson, C. Ray tracing generalized tube primitives: Method and applications. *Computer Graphics Forum*, 38:467–478, 06 2019. doi: 10.1111/cgf.13703.

Quilez, I. Ray-surface intersection functions. https://iquilezles.org/articles/intersectors/.

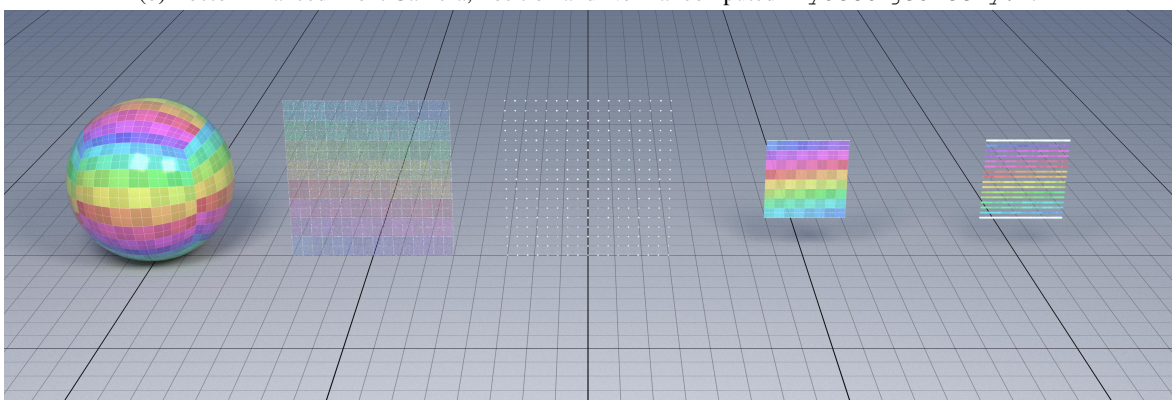
Reshetov, A. Cool patches: A geometric approach to ray/bilinear patch intersections. *Ray Tracing Gems*, 2019.



(a) Yocto Vanilla Front Camera

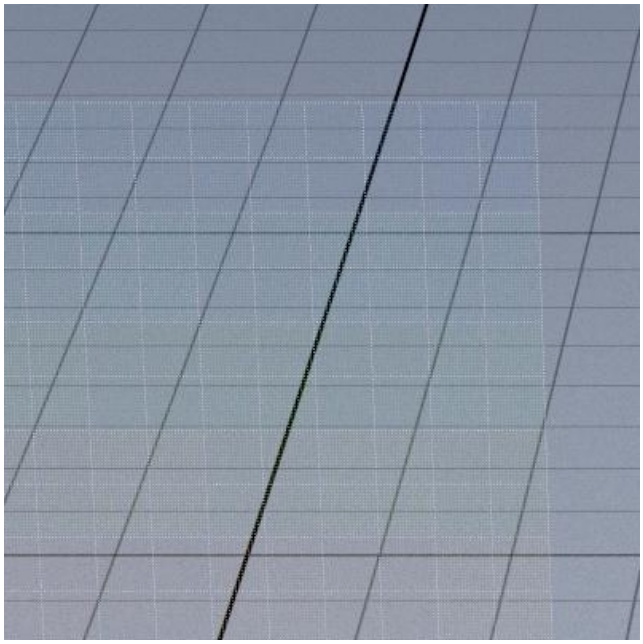


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

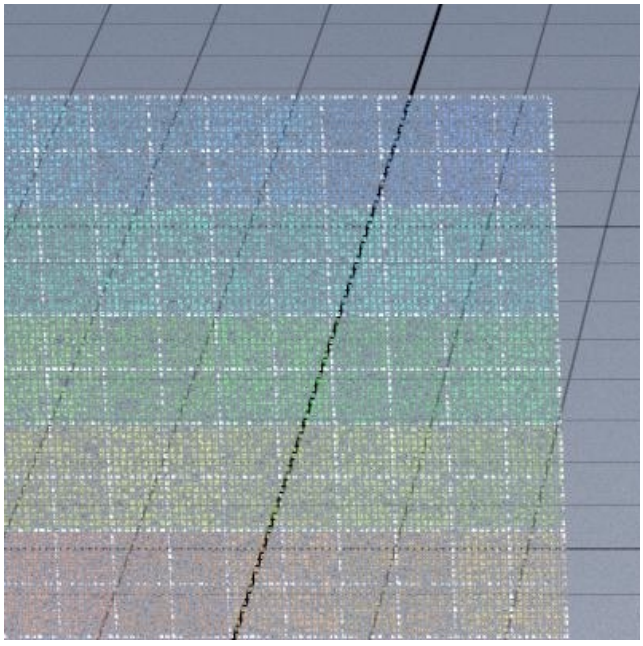


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

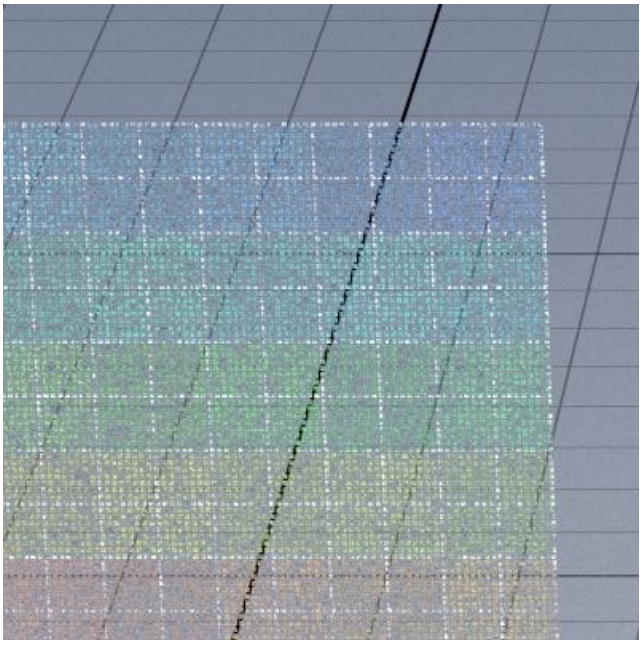
Figure 2. Test of `shapes4` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`.



(a) Yocto Vanilla Front Camera

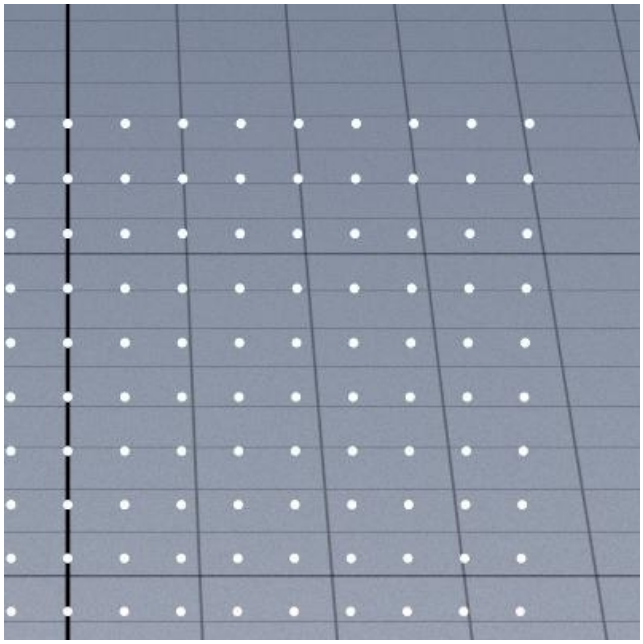


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

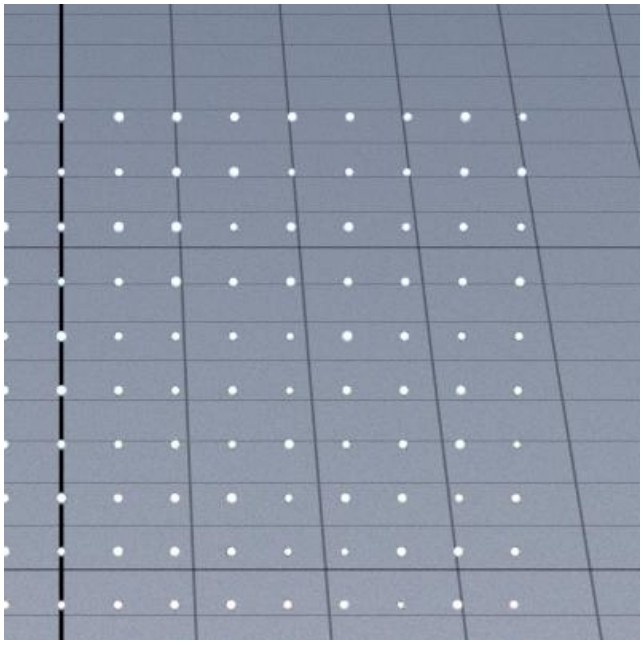


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

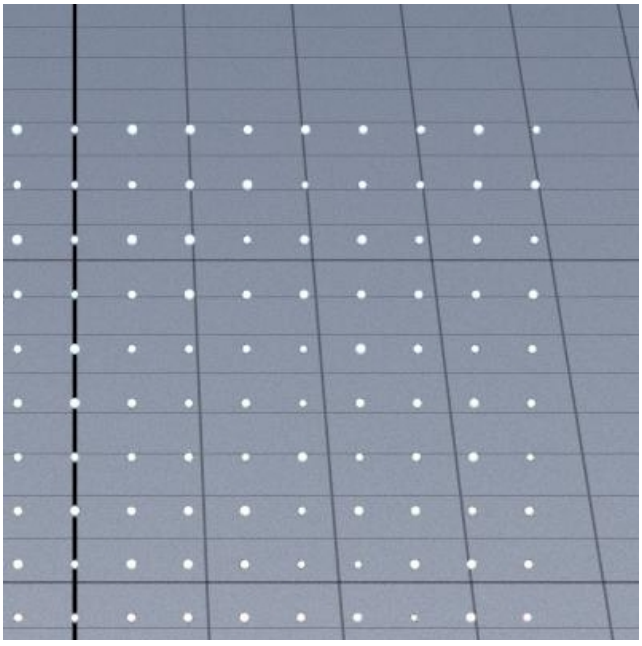
Figure 3. Test of `shapes4` with `sample:256`, `resolution:4060`, `bounce:8`, and `trace_path`, crop on **grid-points**.



(a) Yocto Vanilla Front Camera

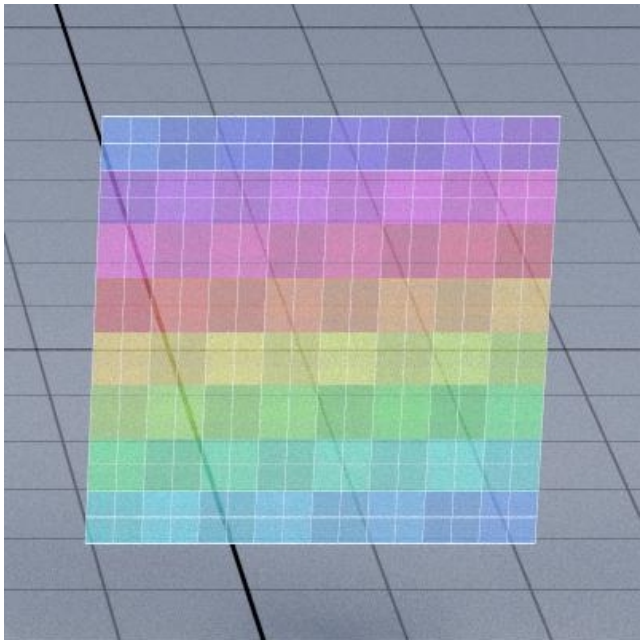


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

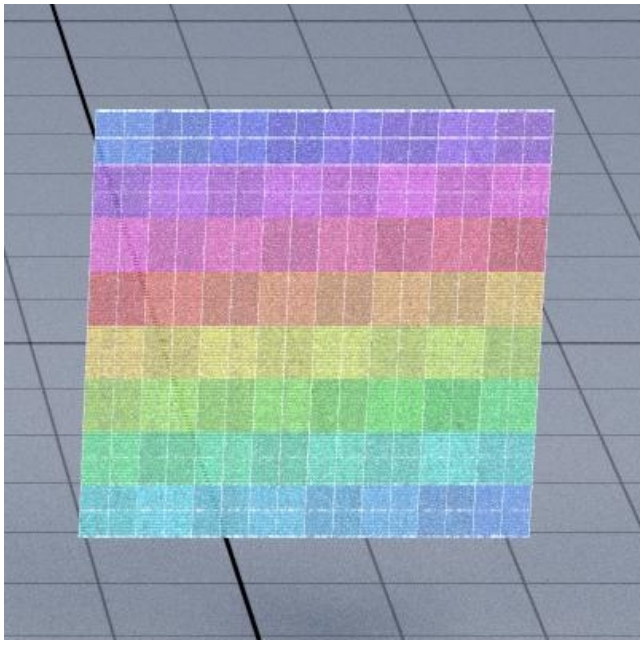


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

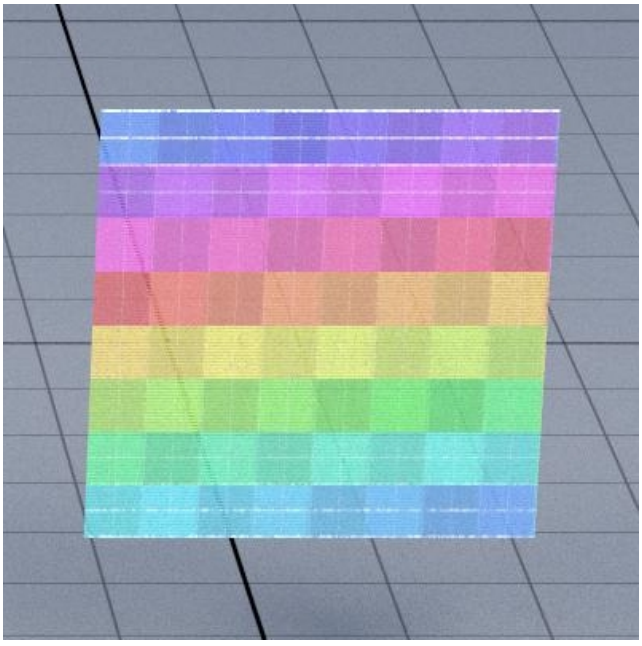
Figure 4. Test of `shapes4` with `sample:256`, `resolution:4060`, `bounce:8`, and `trace_path`, crop on **thick grid-points**.



(a) Yocto Vanilla Front Camera

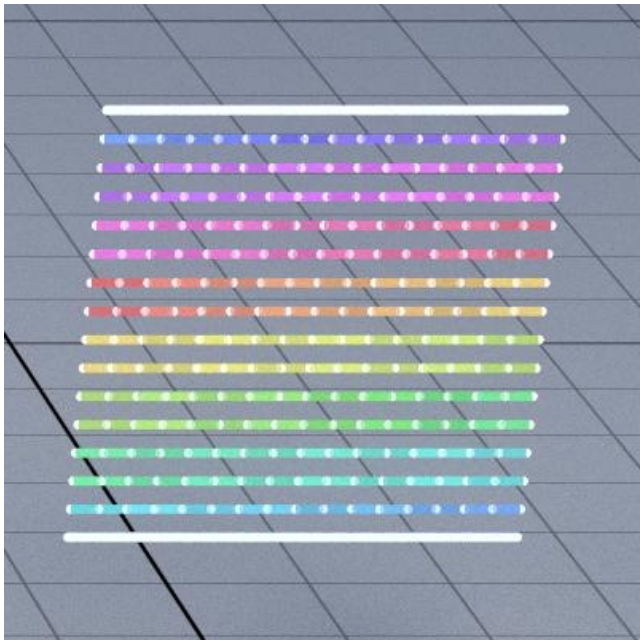


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

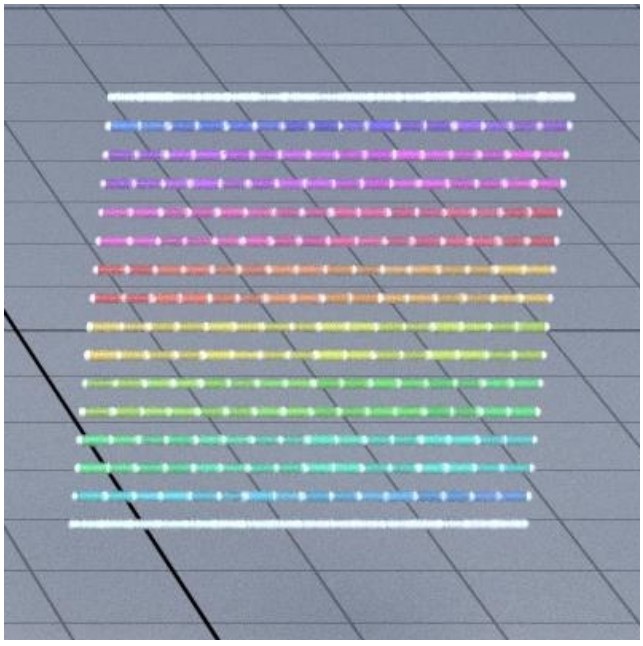


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

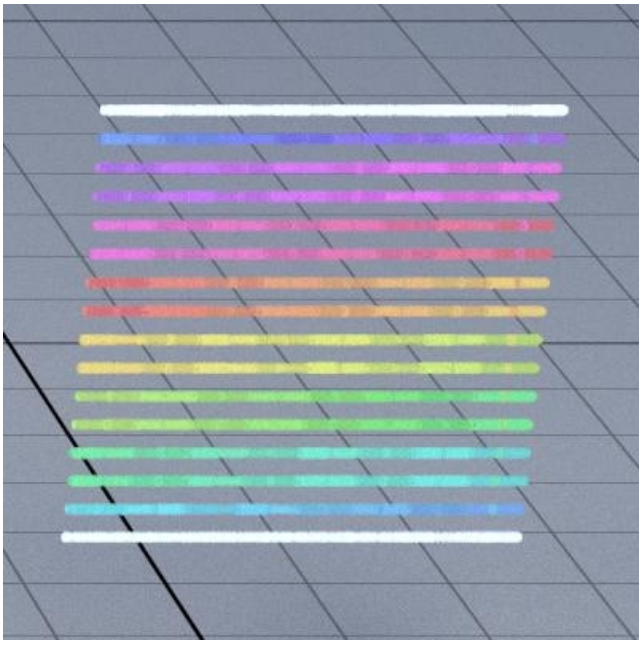
Figure 5. Test of `shapes4` with `sample:256`, `resolution:4060`, `bounce:8`, and `trace_path`, crop on **grid-lines**.



(a) Yocto Vanilla Front Camera

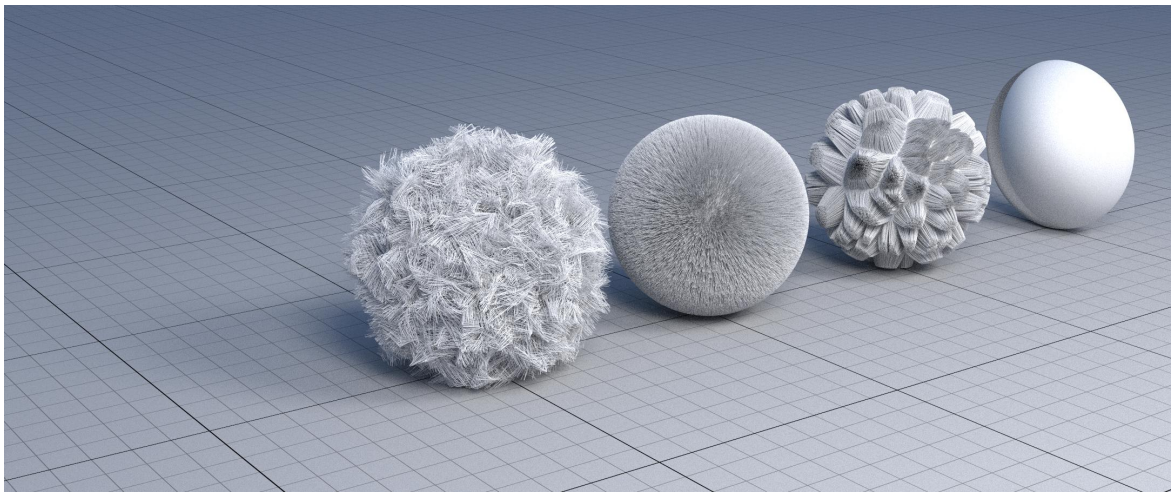


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

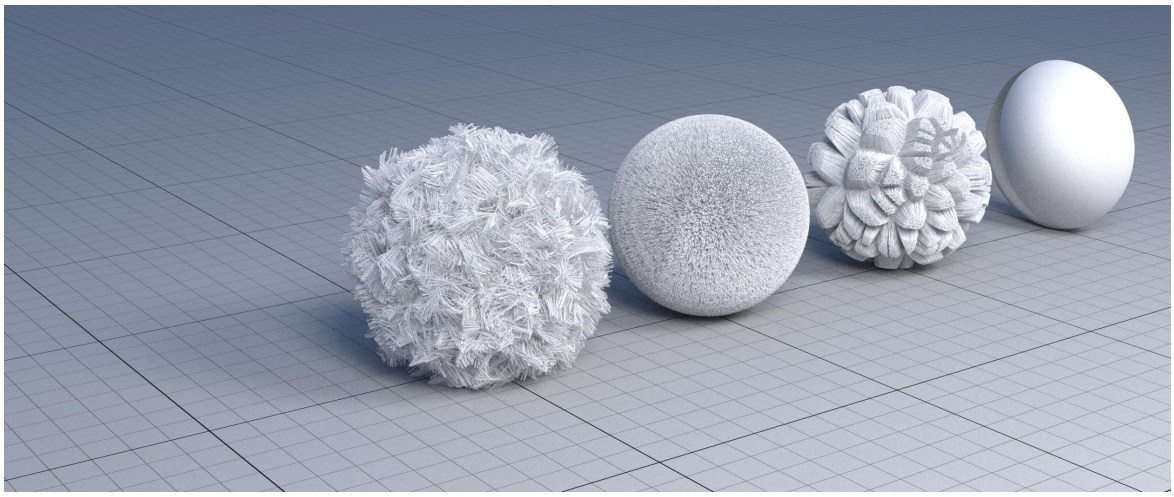


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

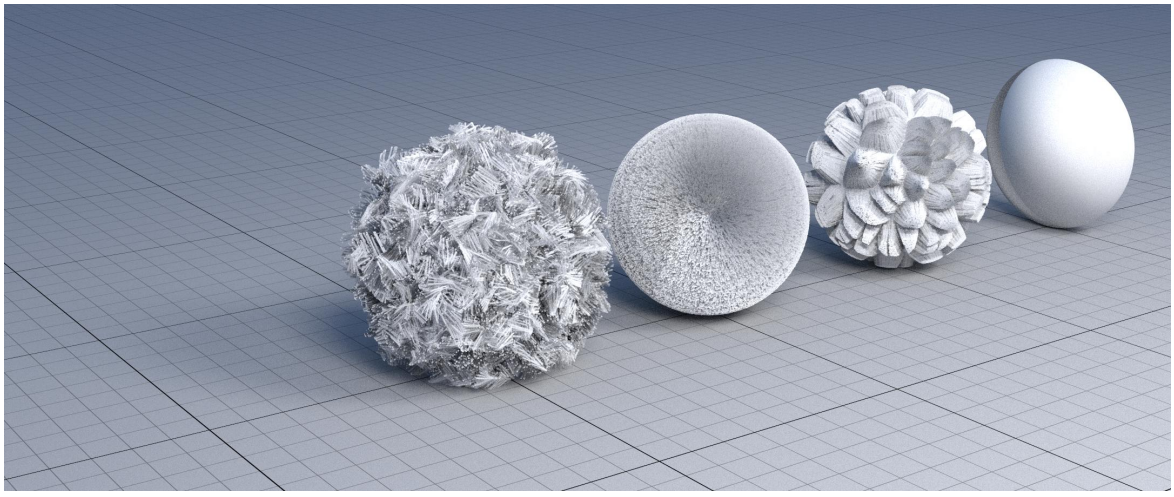
Figure 6. Test of `shapes4` with `sample:256`, `resolution:4060`, `bounce:8`, and `trace_path`, crop on **thick grid-lines**.



(a) Yocto Vanilla Front Camera

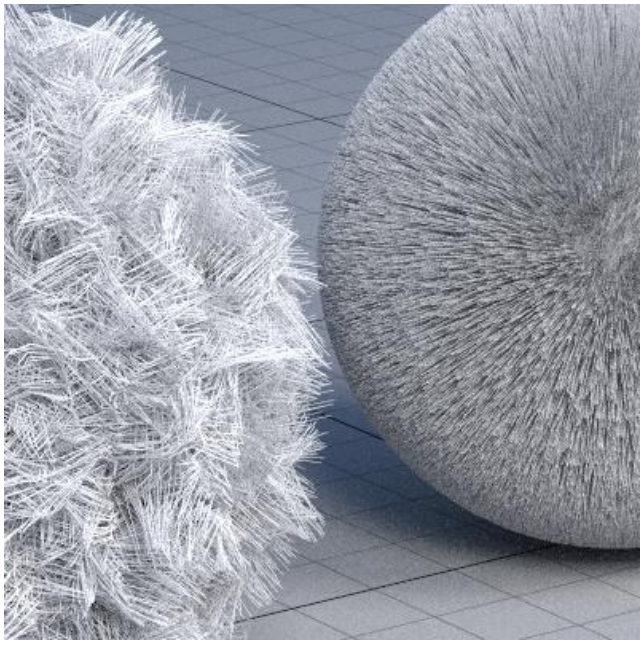


(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.



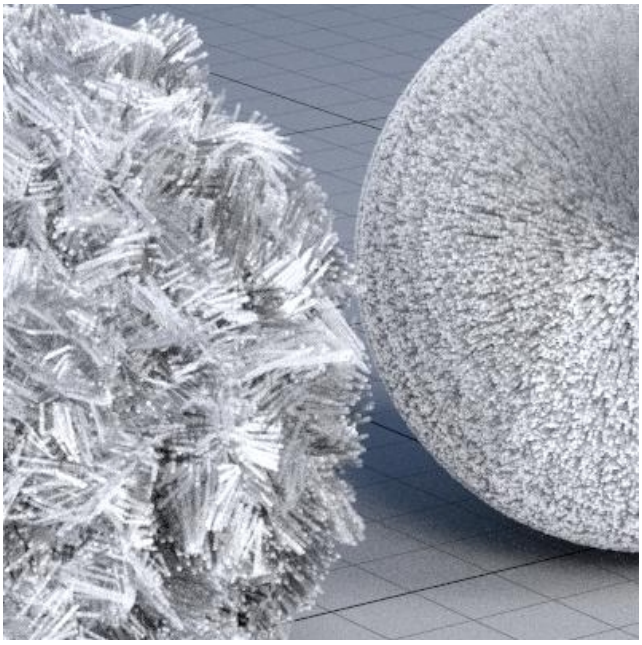
(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

Figure 7. Test of `shapes3` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`.



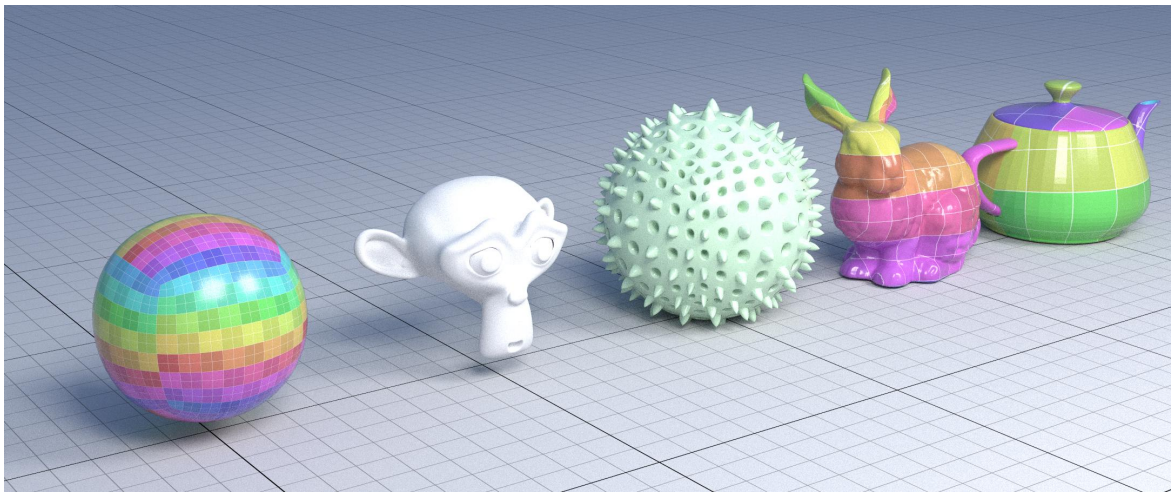
(a) Yocto Vanilla Front Camera

(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_geometry.h`.

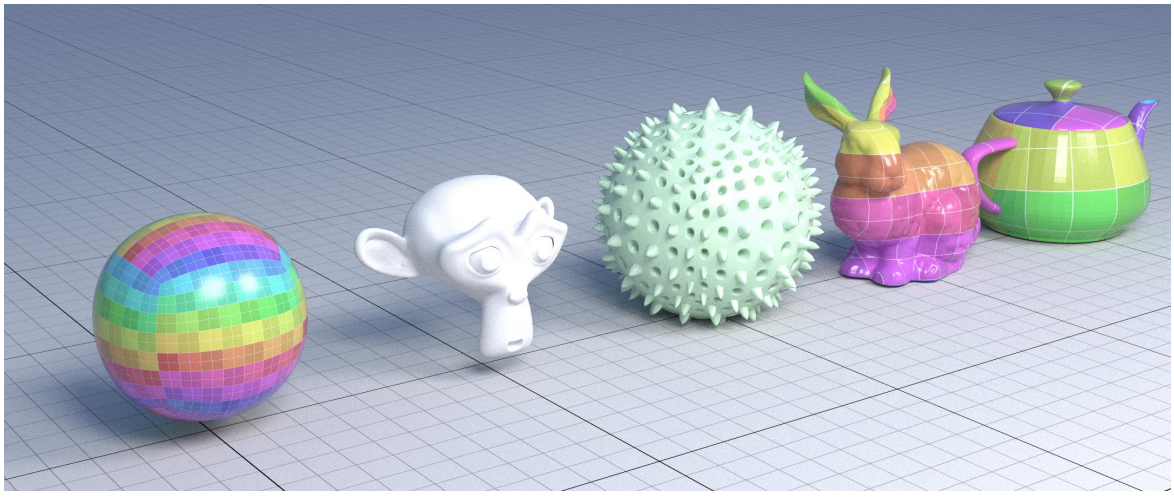


(c) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

Figure 8. Test of `shapes3` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`, cropped.



(a) Yocto Vanilla Front Camera



(b) Yocto Enhanced Front Camera, Position and Normal computed in `yocto_scene.cpp`.

Figure 9. Test of `shapes2` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`.