Intersection Primitives

May 17, 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 reference the work done by Quilez for spheres and cones, and the work of Reshetov for bilinear patches.

2. Related Work

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. In fact, 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 the management of the calculation of the surface normal for the Bilinear Patches, because in order to correctly calculate the surface normal of the Bilinear Patches, it was also necessary to take into account the **type of material** (as can be seen in `eval_shading_normal()` in `yocto_scene.cpp`). This would have involved adding further parameters to the intersection calculation function with the Bilinear Patch primitive, making the code less readable and 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 For the spheres, the approach of the vanilla version of `yocto` has been reused for calculating the UV textures coordinates in `intersect_sphere()` (in `yocto_geometry.h`), using the surface normal and the spherical polar coordinates (cam). In such a way as to be able to carry out the reverse procedure, i.e. the passage from spherical polar coordinates to Cartesian coordinates,

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

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

within the `eval_normal()` and `eval_position()` functions. Finally, to calculate the position, the formula for calculating the normal vector of a sphere was used, i.e. $N = C - P$ (where N represents the normal, C is the center of the sphere and P is the point of intersection). With the normal and center known, the position can be calculated as $P = N + C$.

Rounded Cone As for the Rounded Cone, the UV texture coordinates were calculated using the position and cylindrical polar coordinates in `intersect_cone()` (in `yoto_geometry.h`), and not doing the same way vanilla yocto did with the lines, since in that case the reverse process could not have been carried out.

By doing so, it was possible to get the position through the reverse process in `eval_position()` and `eval_normal()`. For the calculation of the surface normal, **three cases** were handled separately. 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 whether to use vanilla `yocto`'s intersections or use the improved intersections, I introduced checkboxes (Figure 1) that can be selected in the graphical interface (it is also possible to use them as boolean parameters in the command line interface), so that you can also facilitate the performance of the comparison tests. To do this I added three boolean parameters in the `trace_params` structure in `yocto_trace.h`, which are then used in `intersect_shape_bvh()` in `yocto_bvh.cpp`, to select the intersection method.

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).

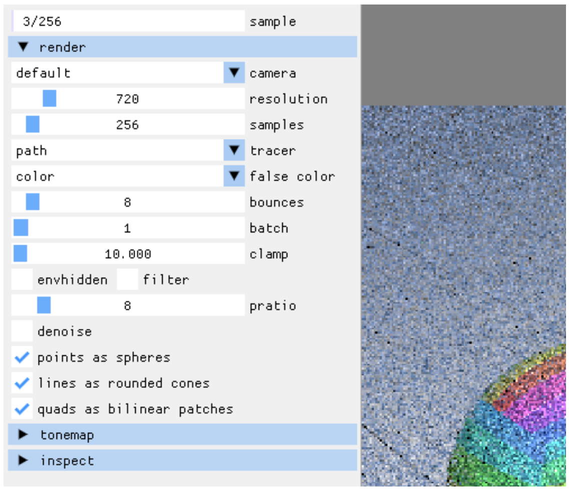
Even for the lines, treated as Rounded Cone, we can say that we have obtained a good improvement in that the shadows results more homogeneous than the light sources (Figure 3), or their color is more evident (Figure 2).

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 4).

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. 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.



```
3/256        sample
▼ render
default      ▼ camera
720          resolution
256          samples
path         ▼ tracer
color        ▼ false color
8            bounces
1            batch
10.000       clamp
envhidden    filter
8            pratio
denoise
✓ points as spheres
✓ lines as rounded cones
✓ quads as bilinear patches
▶ tonemap
▶ inspect
```

Figure 1. Switch to select intersection mode.

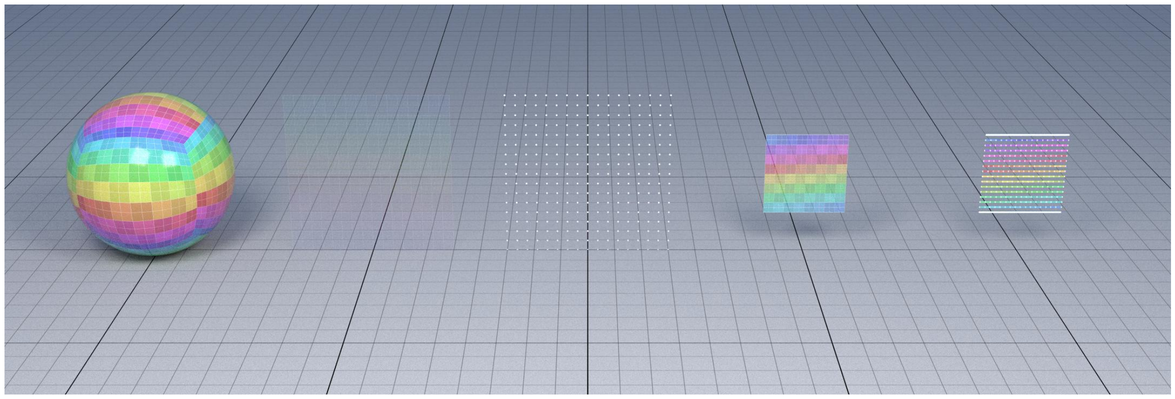
References

Ray tracing primitives. https://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html.

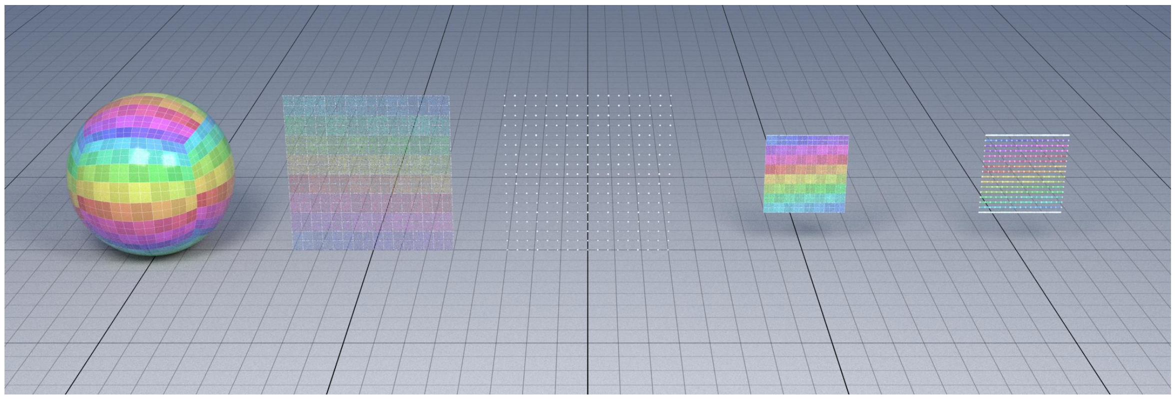
Han, M., Wald, I., Usher, W., Wu, Q., Wang, F., Pascucci, V., Hansen, C., and Johnson, C. Ray tracing generalized tube primitives: Method and applications. *Computer Graphics Forum*, 38:467–478, 06 2019. doi: 10.1111/cgf.13703.

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.

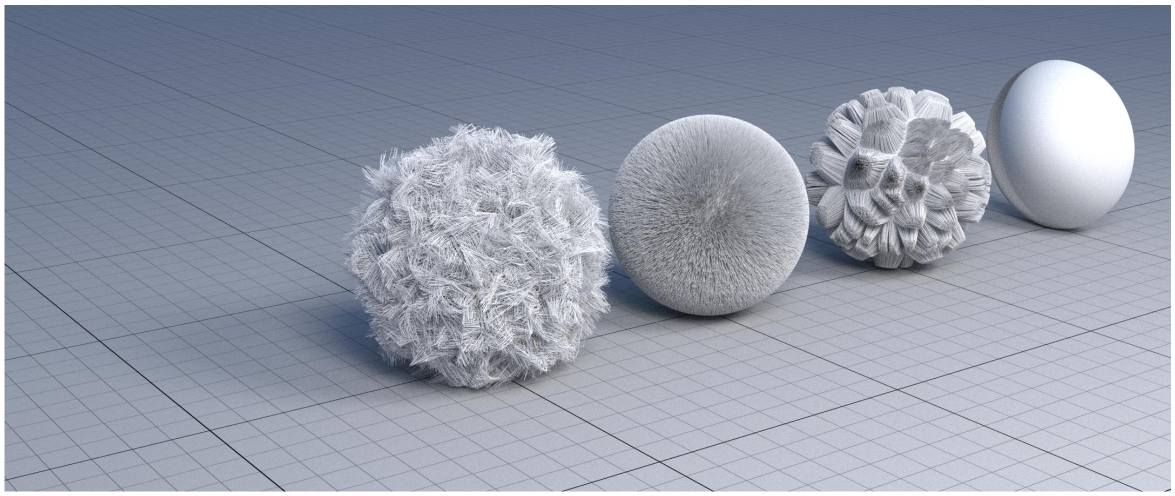


(a) Yocto Vanilla Front Camera

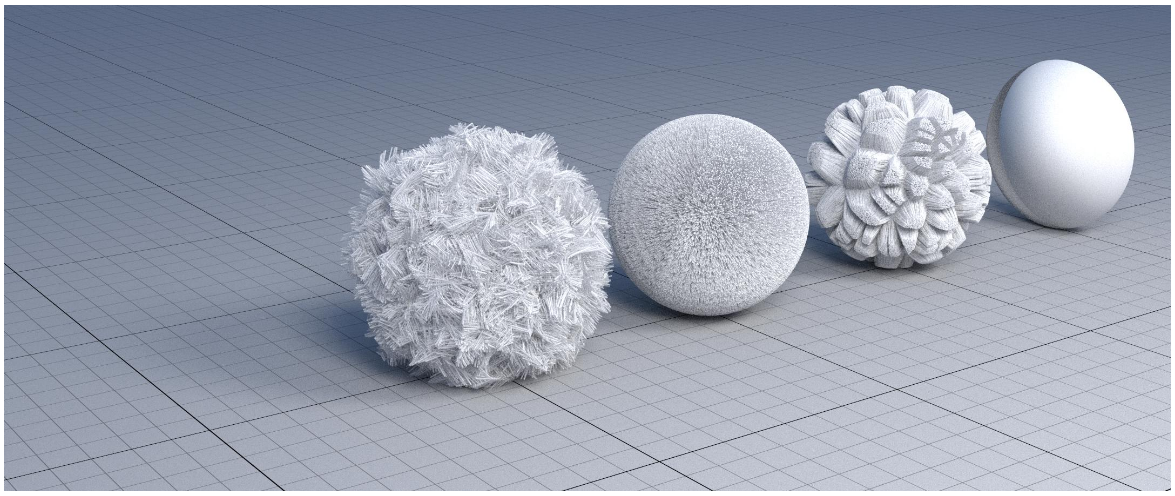


(b) Yocto Enhanced Front Camera

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

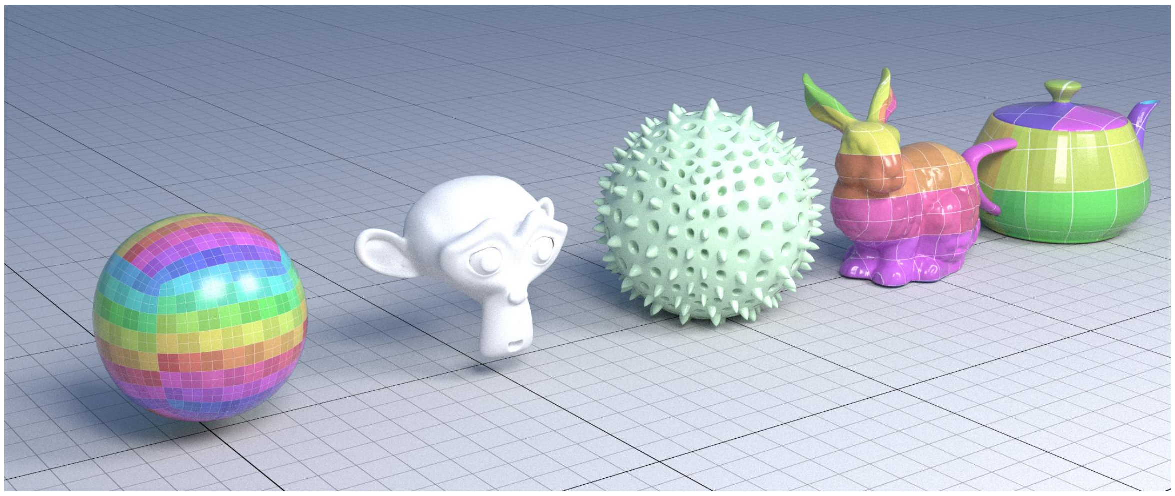


(a) Yocto Vanilla Front Camera

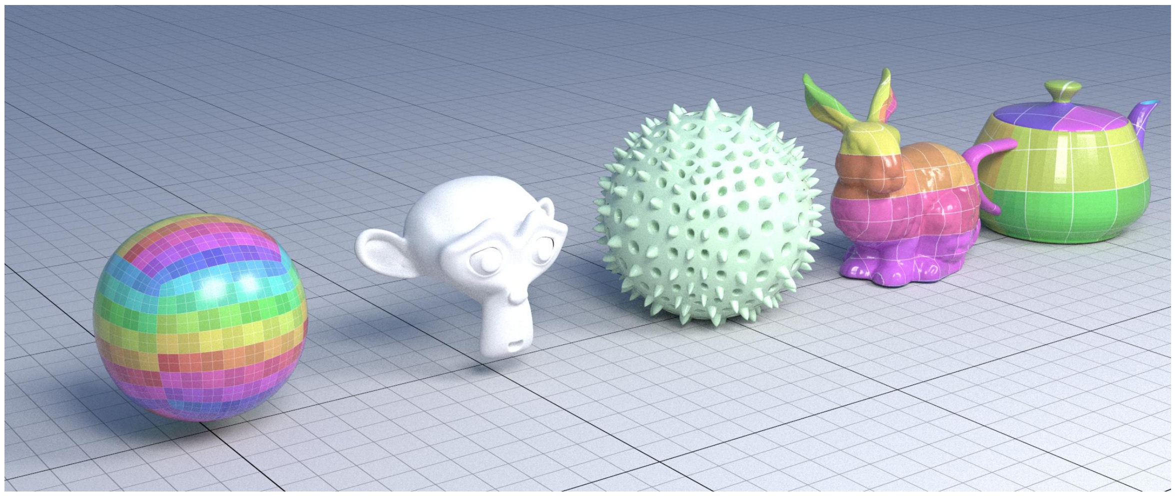


(b) Yocto Enhanced Front Camera

Figure 3. Test of `shapes3` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`.



(a) Yocto Vanilla Front Camera



(b) Yocto Enhanced Front Camera

Figure 4. Test of `shapes2` with `sample:256`, `resolution:2160`, `bounce:8`, and `trace_path`.