

Recursive Ray Tracing in Geometry Shader

Kristóf Ralovich[†] and Milán Magdics[‡]

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest, Hungary

Abstract

We propose an effective method to enable recursive ray tracing triangular scenes using contemporary real-time graphics pipelines of digital computers. So far, general purpose computations such as ray tracing utilized the programmable pixel shader for computation. Lengthy algorithms were subdivided to continue-and-restart-able parts (computing kernels) to enable implementation as multi-pass rendering. Discussing a fundamentally different method, we are representing rays with geometry, maintaining a one-to-one correspondence between rays and point primitives. Exploiting the geometry amplification capability of modern pipeline, we are utilizing the geometry shader to emit multiple secondary rays. Our approach to recursive ray tracing is influenced by stream computing, circulating the data flow without using a stack, employing intermediate pipeline stage to feedback transformed primitives before producing the final color by rasterization of point primitives. An effective implementation employing the uniform grid space subdivision scheme is described.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Raytracing

1. Introduction

Global illumination techniques and thus recursive ray tracing is gaining increasing popularity in real-time image synthesis practice due to widespread availability of consumer level fast parallel digital computer and the algorithmic improvements of the recent years.

Ray shooting maps well and easily to GPUs by the independent nature of rays, and using the graphics pipeline to implement ray tracing had a well known stream computing approach so far: in each rendering pass rasterizing a view-port sized quadrilateral to activate the fragment shader on a regular 2D matrix of pixels where every pass operates on one single ray generation only. A ray generation is for example the first level of shadow rays only (without mixing different

type of rays). Also, e.g. the first bounce of specular reflected rays form an other ray generation.

This paper discusses a fundamentally different configuration of the pipeline, where individual rays are represented by point primitives as opposed to pixels. Recursion required for¹² ray tracing is achieved by feeding back the transformed primitives to the beginning of the pipeline instead of using a large number of textures (render targets) to emulate a stack to store the state of computation. Decomposing⁵ the ray tracing algorithm was historically required because of tight restrictions posed by the GPU on the static and dynamic instruction counts in a shader program. These constraints are not a problem any more, but underlying architectures are still poorly suited for stack memory and thus shading languages can not support recursive function calls directly.

[†] kristof.ralovich@gmail.com

[‡] gumi@inf.elte.hu

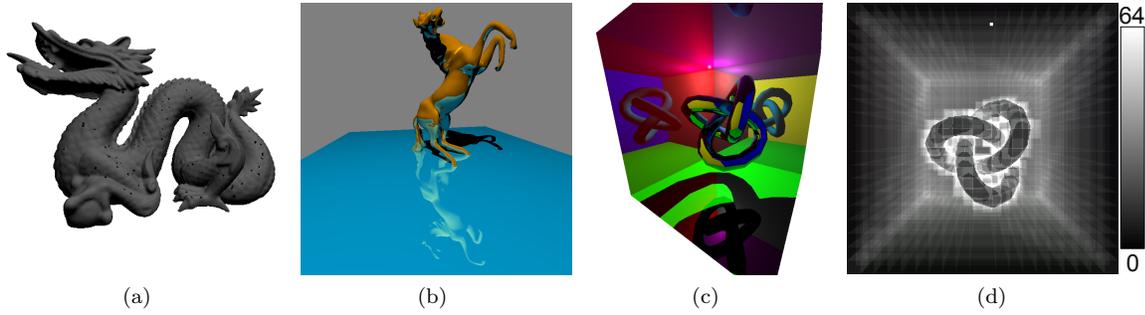


Figure 1: (a)(b)(c) Images generated at interactive rates using the discussed algorithm using geometry shader and transform feedback. Rays are represented in the pipeline with geometry, viewport is 512^2 sized. Images from left to right in respective order: the "Dragon" shadowcasted, the "Horse" scene with specular materials, and the "Torus Knot" scene with reflections and shadows from a single light source. (d) Depicts the cost of finding the intersection for each ray, speed up impact of the employed uniform grid space subdivision is apparent.

2. Previous work

Applying the processing power of programmable GPU to speed up ray tracing motivated many researchers of the field of real time computer graphics.

Carr et. al.¹ discovered that ray-object intersections may be computed in a fragment shader, later others managed to apply a great variety of different techniques such as space subdivision and partitioning (uniform 3D grid^{5,6}, kD -tree^{3,9}, BVH¹¹, etc.) to the GPU to reduce the number of intersections finding the closest visible hit. Szécsi⁸ and Roger et. al.⁷ experimented with ray space hierarchies for logarithmic speedup culling away rays, while Carr et. al.² used geometry images to store the scene GPU memory friendly. The common concept of these methods was using the fragment shader to do the necessary computations.

Szirmay et. al.¹⁰ provides a comprehensive overview of image space methods, practical global illumination, and ray tracing on the GPU.

Recent research shows high performance ray tracing implementations⁴ are possible using the CUDA general purpose parallel programming architecture. CUDA provides direct C language interface to the graphics hardware without special knowledge of programmable graphics pipeline, but we are still focusing on a method layered over graphics APIs because of their wider general availability.

3. Algorithm overview

Let us summarize the structure of our algorithm:

1. Set up two vertex buffers (VB) with enough space.
2. Fill one of the VBs partially with point primitives representing primary rays and bind it as drawing source.

```

struct Hit
{
    vec4 pos; // point primitive
    vec3 orig; // ray origin
    vec3 dir; // ray direction
    vec2 uv; // baryentric hit coords.
    float t; // ray parameter
    int idx; // hit triangle index
    int state; // state vector for
    int type; // inter pass comm.
};
    
```

Listing 1: Data structure holding the hit record.

```

struct PackedHit
{
    vec4 pos;
    vec4 orig_t;
    vec4 dir_idx;
    vec4 uv_state;
};
    
```

Listing 2: Structure encapsulating a ray and corresponding hit record. Encoded as four component floating point vectors. Passed to the pipeline as a point primitive with associated vertex attributes

3. Bind the other VB for stream output destination.
4. Set up the pipeline with the discussed shaders (section 4.1).
5. Rasterize (draw) the point primitives in P number of passes without changing the shaders and without any CPU intervention. A rendering pass consists of the following tasks:
 - a. Calculate intersection in the vertex shader.

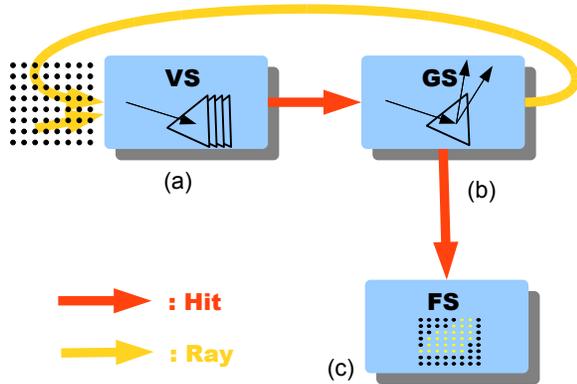


Figure 2: Recursion in the pipeline. Rays are represented by point primitives with additional vertex attributes to hold the hit record. (a) Intersection in the vertex shader. (b) Geometry shader emitting secondary rays and terminating finished rays. (c) Final shading and color compositing in fragment shader.

- b. Depending on material properties, emit secondary rays in the geometry shader.
 - c. Use the stream out stage to feedback transformed points primitives.
 - d. Fragment shader computes color at valid hits.
 - e. Blend the resulting pixel colors together.
6. Swap the VBs. This is called ping-ponging.

4. Algorithm details

The reason we are required to use two vertex buffers is that the source of drawing and the destination of stream output cannot be the same object.

Prior to allocating the VBs, we must know (Figure 4(a)) their maximum sizes: $S_{VB} = w \times h \times (G + G') \times S_V$, where w and h are the dimensions of the viewport, G and G' are the maximum and second largest number of ray generations emitted in each depth of ray tracing recursion, and S_V is the size of a vertex (i.e. the ray structure including a hit record, which is 64 bytes in our case, see code Listing 2). Note that this calculation is not dependent on the number of triangles, and scales with the screen size.

4.1. Pipeline setup

The algorithm is implemented using three stages of the programmable pipeline (Figure 2) and the transform feedback (stream out) functionality. Listings 3, 4, and 5 describe the three shaders in pseudo code.

One could have implemented the functionality of the vertex shader in the geometry shader as well, however

our separation of functionality is geared towards high performance: shorter shaders result in lower branching divergence behavior and thus more coherent memory access patterns.

4.2. Shaders

In the **vertex shader** vertex attributes of point primitives (rays) are loaded from the VB. Ray intersection with the scene is carried out, hit record is updated accordingly and output. As an additional optimization, the ray is checked for intersection against axis aligned bounding box of the scene. If there is no hit, the ray is marked for termination in the subsequent geometry shader.

The output of the vertex processor is considered as the hit record. Depending on the ray state and material properties of the intersection, the ray is terminated (not emitted) or further secondary rays (shadow, reflection) are emitted. Emitted new rays are coupled with a hit record indicating intersections should be calculated in the next vertex shader pass, and to be skipped in the following pixel shader in this pass (because shading requires a valid hit record first). Rays that have been written to the framebuffer in the previous pass are not processed in the **geometry shader** and are terminated (not emitted) early without processing. Transform feedback (Stream Output) is configured to allow receiving multiple output primitives per each input. If there is anything to output, the geometry shader is emitting rays in a fixed local order: first the input ray, later the shadow and finally the specular reflected ray.

Color of rays with valid hit records is calculated employing the Phong shading model and written into the framebuffer in the **pixel shader**, then a special blending operation composites visible color, shadowedness, and the secondary color.

4.3. Rendering

In order to visualize rays in the framebuffer, a viewport sized 2D grid of point primitives is rasterized on a plane perpendicular to the view direction of the virtual camera. The positions of the point primitives are set up so that rasterization assigns them to the pixel centers, which will result in the whole coverage of the screen.

We are exploiting that the number of primitives output from the geometry shader is not needed to be queried back to the CPU to issue a draw call sourcing those vertices.

As stated in section 4.1, the geometry shader has a fixed output order of rays. The graphics pipeline has a

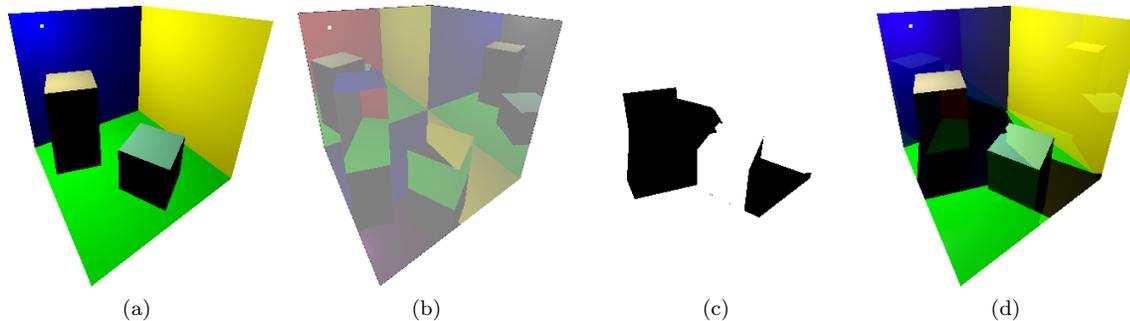


Figure 3: The contribution of eye rays (a), reflected rays (b) and shadow rays (c). The right most (d) image shows the final composited image.

very unique property[†]: primitives hit the framebuffer in the order listed in the VB (if no other indexing is used, like in our case), this is necessary for our algorithm to work, since the processing order of rays must not be changed. Imagine the specular contribution is added to a pixel before processing the previous shadow ray: the shadow ray would amortize the influence of the reflected ray instead of the previous hit.

The P number of rendering passes – required for the correct image – equals to the depth of the ray tracing depth. That is e.g. 2 for primary rays and one bounce (since shadow rays and reflected rays spawned by eye hits are handled together in the same pass). See Figure 4(a).

In order to enable the GPU accessing the scene data, texture memory is employed in a read only manner (in the same way as⁶). We are able to use world space coordinates for the scene, so we do not have to transform rays into object space. Using a uniform space subdivision scheme requires storing the list of referenced objects for each voxel of the grid. This is stored in a RGB 3D texture, where each grid cell is encoded in one texel. The list of referenced objects is stored tightly packed and each texel corresponds to a pointer to actual triangle data. After two levels of indirection, actual triangle data is stored in two other 3D textures both featuring 4 depth slices. Texels with the same (X, Y) coordinates in different Z slices are storing vertex, normal and material information belonging to the same single triangle. Thanks to this “co-location” texture coordinates used for addressing a triangle’s attributes need to be computed only once in the shaders.

[†] Such kind of synchronization can easily be costly to implement in e.g. CUDA.

4.4. Blending

Different ray generations are intermixed in the vertex buffer, and thus final color compositing must be capable of both extinguishing radiance (in case of shadows) and adding radiance (for reflections rays). Considering the case of a single point light source, shadows and one bounce of reflections additional to ray casting, Equation 1 shows one possible[‡] simple compositing setup using fast hardware blending (note that RGB_{src} must be pre-multiplied with the ρ reflectivity to get correct results) in only a single rendering pass.

$$RGB_{out} = (1.0 \times RGB_{src}) + (\alpha_{src} \times RGB_{dst}) \quad (1)$$

$$\alpha_{src} = \begin{cases} 1.0 & \text{for eye rays} \\ 0.0 & \text{for shadow rays} \\ 1.0 & \text{for reflected rays} \end{cases}$$

$$RGB_{src} = \begin{cases} (R_{src}, G_{src}, B_{src}) & \text{for eye rays} \\ (1.0, 1.0, 1.0) & \text{for shadow rays} \\ \rho \cdot (R_{src}, G_{src}, B_{src}) & \text{for reflected rays} \end{cases}$$

This equation lets shadow rays to extinguish the contribution from primary rays. The case of more ray generations and/or light sources require different and a more complicated compositing approach. Ray generations should be sorted and rendered to multiple render targets that blending passes may blend together.

5. Results

Comparison of the distribution of time spent[§] in the shaders is summarized in Table 1. From these results

[‡] An other option is the use of `GL_ARB_color_buffer_float` extension, that widens the possibilities of custom blending, and also provides mechanism to disable clamping of color values before blending and use of negative alpha values.

[§] The presented data is the % of executed instructions distributed between shaders, but since the hardware is run-

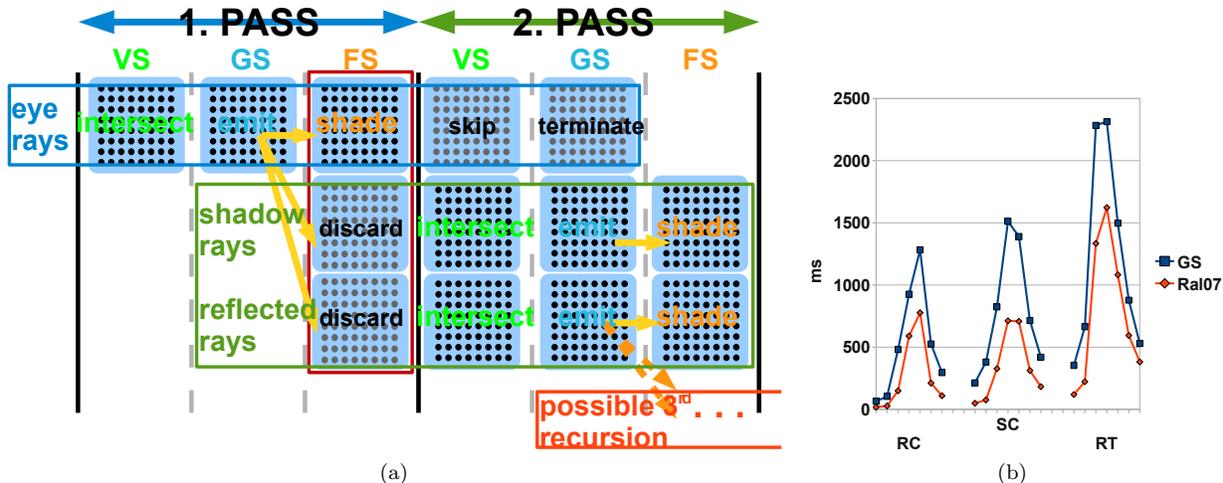


Figure 4: (a) The required number of rendering passes equals to the recursion depth, 2 in the figure (blue, green, and pink frames represent the first, second, and possible third respectively). Each row depicts a different ray generation. Blue areas with dark dots represent the input of stages of a shader program (vertices, fragments) labeled by the task carried out in that shader. Dark red frame shows the maximum size each of the ping-ponging VBs must be capable to accommodate. In the presented case $S_{VB} = 512 \times 512 \times (2 + 1) \times 64 \text{ bytes} = 48 \text{ Mbytes}$. (b) Rendering time (in milliseconds) comparison of our method with previous work⁶. The benchmark environment is the same as for Table 3. We suspect that the 2-3 fold worse performance is due to our using of “fat” (64 byte) primitives.

of naive intersection we conclude, that computation times are limited by vertex shader, that is by the naive intersecting. This is a clear indication that more work may be loaded on the geometry and fragment shaders.

The experiments also showed that the pipeline is compute bound, with this amount of computation the cost of more texture reads may be covered transparently. This leaves us opportunities for improved shading calculations using BRDF sampling.

6. Conclusion and Further Works

4th generation GPUs are built on a unified device architecture. That means the GPU contains only one type of processing unit and that very same unit executes the different types of shaders. This mapping of computations to hardware resources would suggest that it does not matter which type of shader we use to execute the computations and texture reads from, the performance should stay constant. Figure 4(b) shows that this is not true, depicted are our vertex shader[¶] based results compared to the previous results⁶ based

on the fragment shader. We have to conclude that using the vertex shader for intersection calculations is slower than using the pixel shader. This is probably the side effect that we are using “fat” vertices, each point primitive is made up of 64 bytes. We have observed although the geometry shader also has some overhead, the vertex shader even without a geometry shader performs 2-3 times worse^{||} than the fragment shader (see Table 2. for measurements of pure ray casting).

Also moving the light source in the scene incurs high variability in rendering times, this is due that the cost of shadow rays is highly dependent on the position of the light source.

Although efficiency of our setup is apparent, further investigating should be conducted with the current state-of-the-art space subdivision and partitioning methods (kD-tree, BVH, ray hierarchies, etc.) to enable fair comparison with previous techniques using full screen quads.

Extending our work in the future with a best efficiency scene hierarchy would be very interesting.

Our system may be optimized to calculate primary

ning the shader programs on the same unified processors this translates to accurate time measures.

[¶] Shader where the most expensive operation, the intersection test is executed.

^{||} On a Geforce 8600 GPU.

intersections through rasterization of eye rays for the highest possible performance on the graphics hardware. This would only require changing the first pass of the algorithm.

We found that implementing a ray tracer in a graphics API involves a notable runtime overhead. This limitation can easily be overcome once high performance streaming computing software (OpenCL, CUDA) gains more widespread availability.

References

1. Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. *Graphics Hardware*, pages 1–10, 2002.
2. Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. *Proc. Graphics Interface*, pages 203–209, 2006.
3. Tim Foley and Jeremy Sugerma. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.
4. Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, September 2007.
5. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).
6. Kristóf Ralovich. Implementing and analyzing a gpu ray tracer. *Central European Seminar on Computer Graphics*, 2007.
7. David Roger, Ulf Assarsson, and Nicolas Holzschuch. Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques 2007*, pages 99–110. the Eurographics Association, jun 2007.
8. László Szécsi. The hierarchical ray engine. *WSCG*, pages 249–256, 2006.
9. László Szécsi and Kristóf Ralovich. Loose kd-trees on the gpu. *IV. Hungarian Conference on Computer Graphics and Geometry*, pages 94–101, 2007.
10. L. Szirmay-Kalos, L. Szécsi, and M. Sbert. *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, San Rafael, USA, 2008.
11. Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for gpu assisted ray tracing. Master’s thesis, University of Aarhus, 2005.
12. Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

Appendix A: Shader code

```

#define inVS() \
Ray ray=Ray(orig_t.xyz, dir_idx.xyz);\
Hit hit=Hit(uv_state.x, uv_state.y, \
            orig_t.w, int(dir_idx.w));\
int state=int(uv_state.z); \
int type=int(uv_state.w);

#define outVS() \
gl_Position = gl_Vertex; \
orig_t1.xyz = ray.orig; \
orig_t1.w = hit.t; \
dir_idx1.xyz = ray.dir; \
dir_idx1.w = float(hit.idx); \
uv_state1.xy = vec2(hit.u, hit.v); \
uv_state1.z = float(state); \
uv_state1.w = float(type);

void main()
{
inVS();
if(state == 0){//generate eye rays
ray=Ray(cameraPos, normalize(vec3(
gl_Vertex.x, gl_Vertex.y, -1.0)*rot3));
hit.t = INF;
hit.idx = -1;
state = 1;
type = 0;
hit = intersect_grid(ray, hit.t);
}
#if defined(SHADOWS) || defined(
RECURSION)
else if(state == 1) {
hit = intersect_grid(ray, hit.t);
}
#endif
outVS(); // else state == 3
}

```

Listing 3: GLSL code for the vertex shader.

```

void main()
{
    inGS();

    if(state > 1)
        return;

    if(hit.idx == -1)
        return;

    emitPassThrough();

    if(type != 0 || emitNoMore>0)
        return;

    #if defined(SHADOWS) || defined(
    RECURSION)
        lookupNormal(hit, hitN);
        vec3 hitP = ray.orig + ray.dir*hit.t
            + hitN*EPSILON;
    #endif
    #ifndef SHADOWS
        vec3 toLight = lightPos - hitP;
        float lightDist = length(toLight);
        Ray shadowRay = Ray(hitP, toLight/
            lightDist);
        Hit shadowHit = Hit(0.0, 0.0,
            lightDist, -1);
        state = 1;
        type = 1;

        emitShadowRay();
    #endif
    #ifndef RECURSION
        Ray reflRay = Ray(hitP, reflect(ray.
            dir, hitN));
        Hit reflHit =Hit(0.0, 0.0, INF, -1);
        state = 1; // intersect in
            next pass, FS discard in this
            pass
        type = -1;

        emitReflRay();
    #endif
}

```

Listing 4: GLSL code for the geometry shader.

```

void main()
{
    Ray ray=Ray(orig_t2.xyz,dir_idx2.xyz);
    Hit hit=Hit(uv_state2.x,uv_state2.y,
        orig_t2.w,int(dir_idx2.w));
    int state = int(uv_state2.z);
    int type = int(uv_state2.w);
}

```

```

if(state < 3)
    discard;
if(type == 0)
{
    Ray eyeRay = ray;
    Hit eyeHit = hit;
    if(eyeHit.idx == -1)
    {
        gl_FragColor = vec4(backgroundColor.
            rgb, 0.0);
        return;
    }
    vec3 eyeHitPosition = eyeRay.orig +
        eyeRay.dir * eyeHit.t;
    vec3 lightVec = lightPos -
        eyeHitPosition;
    lookupNormal(eyeHit, N);
    vec3 L = normalize(lightVec);
    float NdotL = max(dot(N, L), 0.0);
    vec3 diffuse = lookupTriangleColor(
        eyeHit.idx); // material color of
        the visible point
    gl_FragColor = vec4(diffuse * NdotL,
        1.0);
    return;
}
#ifndef SHADOWS
if(type > 0)
{
    Hit shadowHit = hit;
    if(shadowHit.idx == -1)
        discard;
    gl_FragColor = vec4(-1,-1,-1, 0.0);
    return;
}
#endif
#ifndef RECURSION
{ // else type < 0
    Ray reflRay = ray;
    Hit reflHit = hit;
    if(reflHit.idx == -1)
        discard;
    vec3 reflHitPosition = reflRay.orig +
        reflRay.dir * reflHit.t;
    vec3 lightVec = lightPos -
        reflHitPosition;
    lookupNormal(reflHit, N);
    vec3 L = normalize(lightVec);
    float NdotL = max(dot(N, L), 0.0);
    vec3 diffuse = lookupTriangleColor(
        reflHit.idx);
    gl_FragColor = vec4(diffuse*NdotL
        *0.25 1.0);
}
#endif
}

```

Listing 5: GLSL code for the fragment shader.

Scene (# Δ)	Vertex Shader (%)	Geometry Shader (%)	Fragment Shader (%)
room3 (12)	67.43 / 48.07 / 54.73	1.14 / 5.15 / 5.05	31.51 / 46.79 / 40.21
Cornell Box (36)	85.34 / 77.43 / 72.70	0.50 / 2.47 / 2.68	14.14 / 20.16 / 24.62
Knight in Box (646) (198 animated frames)	99.20 / 98.51 / NA	0.03 / 0.18 / NA	0.76 / 1.32 / NA

Table 1: Computation time distribution between shaders using naive intersection testing. Numbers are for ray casting, shadow casting and shadow casting with single reflections with naive intersection testing (not using the uniform grid). All images are ray traced in 512×512 viewport, all rays are intersecting the scene. GPU instrumentation details are as reported by NVIDIA PerfHUD on a single Geforce 260 GTX, as an average of 50 independent experiments.

	Our method with passthrough GS	Our method without Geometry Shader	⁶ using Fragment Shader
Rays / second (million)	3.322,484	4.890,746	10.485,760
Ray shooting time (ms)	78.9	53.6	25.0

Table 2: Pure ray shooting performance of different setups of the graphics pipeline. All images are ray traced in 512×512 viewport, all rays are intersecting the Torus Knot scene consisting of 1024 triangles. GPU time was computed asynchronously as reported by GL_EXT_timer_query on a single Geforce 8600, as an average of 50 independent experiments (by removing the best and worst from 52 independent experiments).

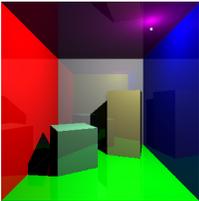
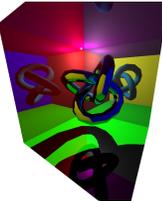
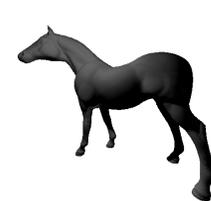
	Cornell Box	Cornell Knot	Dragon	Fairy Forest	Happy Buddha	Horse	Stanford Bunny	
								
# Δ	36	1,024	871,414	174,117	1,087,716	96,966	69,451	
Our Method	RC	64.4	104	481	926	1283	526	295
	SC	213	381	824	1512	1390	714	419
	RT	353	664	2284	2314	1497	878	529
Previous ⁶	RC	17.5	26.8	149	590	777	211	109
	SC	48.5	75.3	327	712	708	312	182
	RT	119	222	1334	1623	1082	594	381

Table 3: Ray tracing performance in milliseconds. Numbers are for ray casting (RC), shadow casting (SC) and shadow casting with single reflections (RT). All images are ray traced in 512×512 viewport. GPU time was computed asynchronously as reported by GL_EXT_timer_query on a single Geforce 8600, as an average of 50 independent experiments (by removing the best and worst from 52 independent experiments).