5
Ray Tracing on the GPU

In this chapter we review the scene data layout on the GPU, as well as our proposed ray tracing implementation. Firstly, we identify the common routines used to trace rays using a Uniform Grid inside the GPU. Afterwards, we describe our optimized ray tracing implementation using the graphics hardware.

5.1
Data Layout

Scene data corresponding to the current frame is stored inside the GPU as a set of 2D textures, as follows:

1. Triangle vertices $\boxed{x_0,\ y_0,\ z_0 \mid x_1,\ y_1,\ z_1 \mid \ldots}$
2. Per-vertex normals $\boxed{n_{x0},\ n_{y0},\ n_{z0} \mid n_{x1},\ n_{y1},\ n_{z1} \mid \ldots}$
3. Per-vertex texture coordinates $\boxed{s_0,\ t_0 \mid s_1,\ t_1 \mid \ldots}$
4. Per-triangle materials $\boxed{d_{r0},\ d_{g0},\ d_{b0},\ tex_{id0} \mid d_{r1},\ d_{g1},\ d_{b1},\ tex_{id1} \mid \ldots}$
5. Grid indices $\boxed{start_0,\ size_0 \mid start_1,\ size_1 \mid \ldots}$
6. Grid data $\boxed{cell_{id0},\ prim_{id0} \mid cell_{id0},\ prim_{id1} \mid \ldots}$

The first four textures contain geometry and shading information that must be updated each animation frame. Material information include the diffuse colors of each triangle, as well as a texture ID if there are textured triangles. The last two textures correspond to the Uniform Grid structure. Whenever there is movement in the scene, these textures are updated by the grid construction procedure, as described in the previous chapter.

5.2 Common Routines

It is possible to implement the entire ray-tracing procedure inside the GPU using either the CUDA API or GLSL shaders. Our tests have shown that a CUDA implementation is significantly slower than an equivalent GLSL one, even after taking advantage of coherent memory accesses in the graphics card.

A good reason for this result is that the GLSL implementation is rather simple: a full-screen quadrilateral is used to trigger a fragment shader, which in turn performs the necessary computations and outputs the pixel values directly to the screen. In addition, all information can be efficiently retrieved from textures exploiting cached read operations.

In order to trace rays using a Uniform Grid inside the GPU, a few common tasks must be performed. First of all, it is necessary to upload the current view information, such as camera position and orientation. These values are used to setup the primary rays. In addition, common ray-traversal and intersection routines can be formulated. Finally, a shading procedure must be able to compute correct illumination, considering shadows and reflections.

Figure 5.1 describes a conceptual ray-tracing algorithm that integrates all these steps in order to render each new frame. Notice how each routine interacts with different scene and grid data, while also being re-used for tracing secondary rays such as shadow and reflection ones.

This conceptual algorithm forms the basis of our optimized procedure, as described in Section 5.3. Before that, we present the following common routines that will be used in our final implementation.

5.2.1 Viewing Information

At each rendering frame, the GPU routines need to obtain the current view description, namely its origin and direction. Since we use a pinhole camera model, the ray origin is the same for all primary rays. Therefore, this information is sent as an uniform variable.

To compute the ray directions, the implementation exploits a pre-existing hardware functionality. From the pinhole camera model, all ray directions can be linearly interpolated in screen space. This means that we can use the attributes of each of the four quadrilateral vertices to interpolate the direction information. This can be done automatically by the hardware rasterizer.

In other words, the rendering method sends the ray directions of the lower left, lower right, upper right and upper left rays each to its corresponding

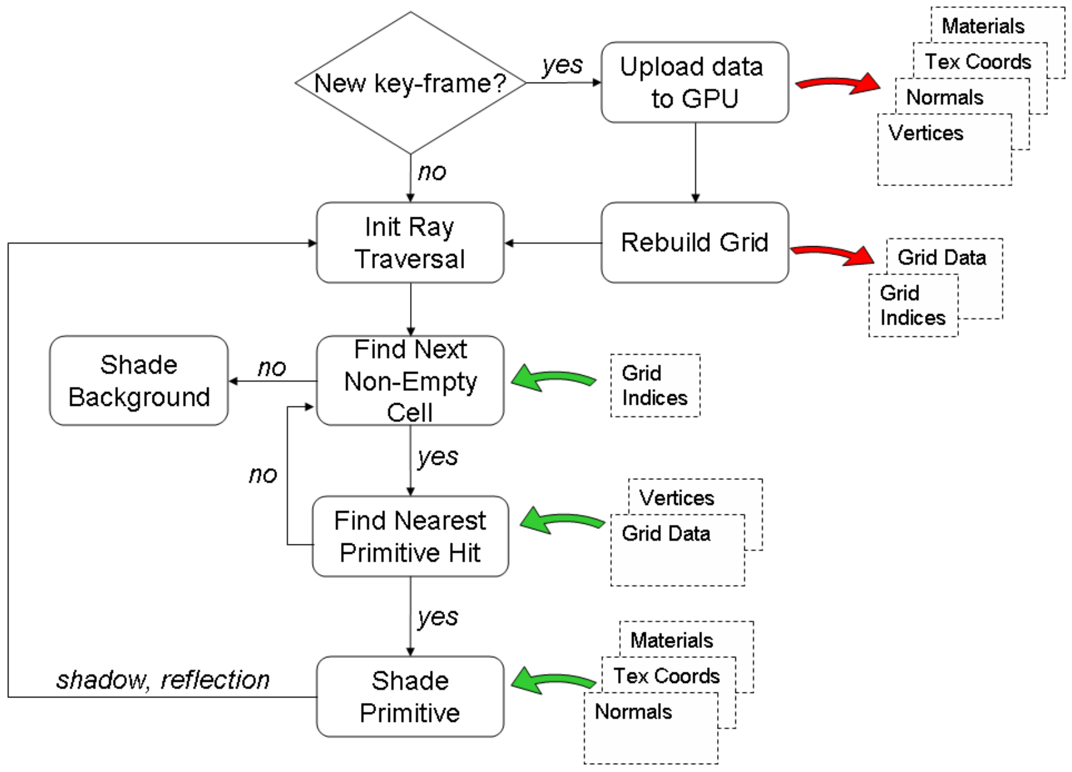

Figure 5.1: Conceptual ray-tracing algorithm. Texture read and write operations are highlighted in green and red, respectively.

full screen quadrilateral vertex. The resulting directions will be automatically interpolated to the fragment shader. The traversal and intersection routines do not require the directions to be normalized, so this technique is effectively free.

5.2.2 Uniform Grid Traversal

Before the main traversal loop, the algorithm pre-computes all the required information for the 3D-DDA traversal. These include the starting cell coordinates, the constant delta values from Section 3.2 and the out limits to stop traversal if the ray exits the bounding box.

These pre-computed values are used to traverse the ray through the grid structure. At each step, the algorithm reads the primitives contained by the current cell, converting its 3D coordinates to a 2D texture address. The procedure uses the grid index texture to determine the starting cell indices as well as its size, which in turn determine where the current cell's primitive list is located in the main grid texture.

If no ray-primitive intersection is found, traversal continues until the ray exits the grid structure without hitting any triangles. On the other hand, if an intersection is found the procedure stores the hit information and returns.

5.2.3 Ray-Triangle Intersection

Since the GPU has limited resources such as total available memory and its bandwidth, we have implemented the ray-primitive intersection routine based on the algorithm by Möller-Trumbore [Möller and Trumbore 1997]. Its main advantage is that no additional information is required per triangle, besides its three vertices. Moreover, this routine performs nearly as fast as other approaches, which in turn require pre-computed coefficients for each triangle. For instance, an optimized algorithm that uses an additional 48 bytes per triangle performs at most 20% faster [Wald 2004].

The main principle behind the intersection algorithm is to apply a transformation to the origin of the ray, so that the resulting vector contains the distance t to the intersection coordinates (u, v) inside the triangle. A point $T(u, v)$ on a triangle is given by:

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2 \tag{5-1}$$

where (u, v) are the barycentric coordinates, which must fulfill $u \geq 0$, $v \geq 0$, $u + v \leq 1$. Computing the intersection between the $Ray(t)$ and the triangle $T(u, v)$ is equivalent to:

$$O + tD = (1-u-v)V_0 + uV_1 + vV_2 \tag{5-2}$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{5-3}$$

This means the barycentric coordinates (u, v) and the intersection distance t can be found by solving the linear system of equations above. The implementation uses Cramer's rule to reduce the solution to a few multiplications and vector products, reusing as many common factors as possible.

5.3 Optimized Implementation

There are fundamentally two approaches to implementing the ray tracing algorithm with GLSL. The first is to use a single fragment shader to encode the entire ray setup, traversal, intersection and shading routines. The other is to break the procedure into several rendering passes. We have verified experimentally that separating the traversal and intersection from the shading routines can perform up to two times faster than a single shader approach.

Therefore, our proposed ray tracing implementation on the GPU can be summarized in three main steps, each performed by a different fragment shader:

1. Primary ray traversal and intersection
2. Shadow ray traversal and intersection
3. Shading computations

The following subsections describe the computations done by each step, including their respective input and output information.

5.3.1 Primary Ray Traversal and Intersection

After initializing the primary rays according to Subsection 5.2.1, the shader in Step 1 first checks for intersection against the grid bounding box. If none is found, the shader outputs invalid values. In the other case, the hit position is evaluated and the first cell to be traversed is obtained. The algorithm then pre-computes all the required information for the 3D-DDA traversal, which is then performed according to Subsection 5.2.2.

The output of Step 1 is a texture that contains, for each texel, the following hit information: triangle ID, barycentric coordinates (u, v) and the hit distance. If no hit was found, the shader writes invalid values.

5.3.2 Shadow Ray Traversal and Intersection

The shader in Step 2 uses the hit information from Step 1 to compute the origin and direction of the shadow rays, tracing them using an optimized procedure similar to the ones described in Subsections 5.2.2 and 5.2.3.

The fundamental difference is that to determine if a point is in shadow, all we need is to find any intersection along the shadow ray. In this case, the hit distance is irrelevant. In addition, there is no need to check against the grid bounding box, as we know for sure the ray origin is already inside of it.

Shadow rays are cast from the primary hit position towards a global point light. The output of Step 2 is a hit information similar to Step 1, with a difference: the shader changes the triangle ID to a negative value, identifying a successful shadow hit. This information will be used in Step 3 to perform the correct shading computations. If no shadow hit was found, the shader writes the same hit information obtained from the first step.

5.3.3 Shading Computations

Step 3 reads the hit information from Step 2 to recover the triangle ID, barycentric coordinates (u, v) and hit distance. The triangle ID is used to recover the necessary triangle information (normals, texture coordinates and materials) to perform all shading computations. The barycentric coordinates are used to interpolate per-vertex attributes such as the normal and texture coordinates.

Finally, the hit distance is used to evaluate the hit position in space, which is necessary to compute the Phong illumination with the Lambertian reflectance model. If the hit is in shadow, no specular is computed and the final color is divided by a constant factor.

5.4 Enabling Reflections

One of the main advantages of ray tracing, as mentioned in Chapter 1, is the possibility to evaluate global radiance information simply by tracing additional rays. We confirm this observation by re-using the aforementioned algorithm to shade reflective materials. Our strategy consists in another rendering pass, performing the same Steps 1, 2 and 3 but now for reflection rays. The final color values are modulated with the original ones, using the OpenGL blend operation.

A uniform variable is used as a flag to inform each shader if reflections are being rendered. In this case, Step 1 does not initialize the rays using the view information, but reads the primary hits previously computed in the same frame. The shader then computes the reflection ray direction using the surface normal at the primary hit position. Traversal and intersection is performed, and the secondary hits are written in an off-screen framebuffer.

Likewise, Step 2 reads the secondary hits to compute shadow information for the surface reflections. As before, the output indicates whether the hits are in shadow or not. Finally, Step 3 reads these new hits and performs the same shading operations as with the primary rays.

Clearly, this iterative process could be repeated, each time shading another level of reflections. For testing purposes, we have limited our implementation to a single level.