3
Solution Overview

Before analyzing our ray-tracing implementation, we present the basic concepts involved in building Uniform Grids and how to use these structures to accelerate ray tracing. We then correlate these techniques to the paradigm of GPU programming, proposing a complete solution fully implemented inside the graphics hardware.

3.1
Uniform Grid Structure

The Uniform Grid is a regular spatial subdivision structure. The axis aligned bounding box of the entire scene is subdivided into equally sized cells along each of the three main axis X, Y and Z. The number of cells in each axis, and consequently its size, can be different. The important characteristic is that all cells have the same width, height and depth. Each cell stores a reference to the primitives it contains.

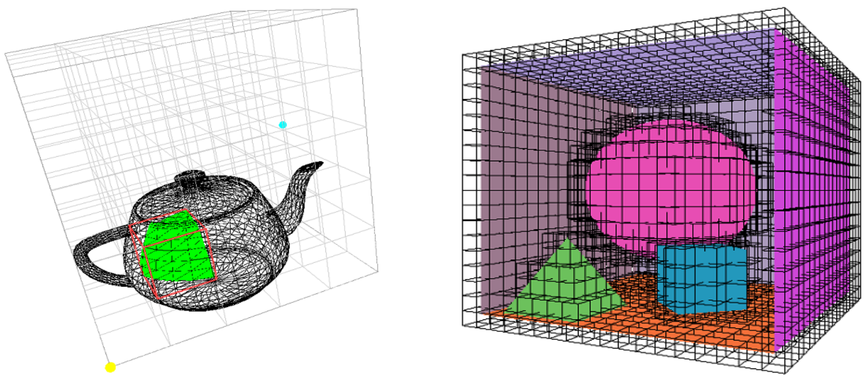


Figure 3.1: Examples of Uniform Grids.

To obtain best ray-tracing performance, the deciding factor is to choose a good grid resolution. The typical heuristic to determine the number of cells in each dimension attempts to consider both the complexity of the scene and the size it occupies in space [Wald et al. 2006]. In other words, a fairly complex scene would demand a more refined grid. Similarly, a scene that occupies a very large space could be represented by a more sparse grid. This leads to Equations 3-1, as follows:

$$N_x = d_x \sqrt[3]{\frac{kP}{V}} \qquad N_y = d_y \sqrt[3]{\frac{kP}{V}} \qquad N_z = d_z \sqrt[3]{\frac{kP}{V}} \tag{3-1}$$

N_x, N_y and N_z are the number of grid cells in each dimension

P is the total number of primitives in the scene

V is the total volume of the grid

d_x, d_y and d_z form the diagonal of the grid

k is a user-defined constant to determine a more dense or sparse grid

The main step during grid construction is determining for each primitive which cells it overlaps. A direct approach is to use the primitive's axis-aligned bounding box to find the overlapping cells. A more refined one would compute the actual primitive-cell intersections (see Figure 3.2). Either way, our goal is to implement this procedure entirely inside the GPU. This has two main benefits: it avoids costly memory transfers from the CPU to the GPU, and it opens the opportunity to explore the parallel computing power of the graphics hardware.

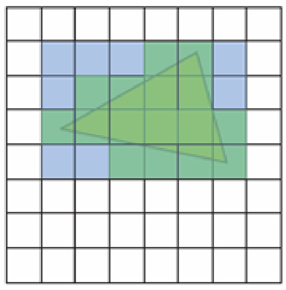


Figure 3.2: In blue, cells overlapped only by the triangle's AABB. In green, cells overlapped by the actual primitive.

A typical grid representation stores a list of primitives contained by each cell. Consecutive lists are arranged contiguously in memory. In practical use, the grid structure must support random access during ray traversal. This means that it is necessary to determine all primitives contained by a given cell using only its ID. Thus, in addition to the actual grid data we need to build an index to translate the cell ID into the beginning of its corresponding primitive list, as shown in Figure 3.3.

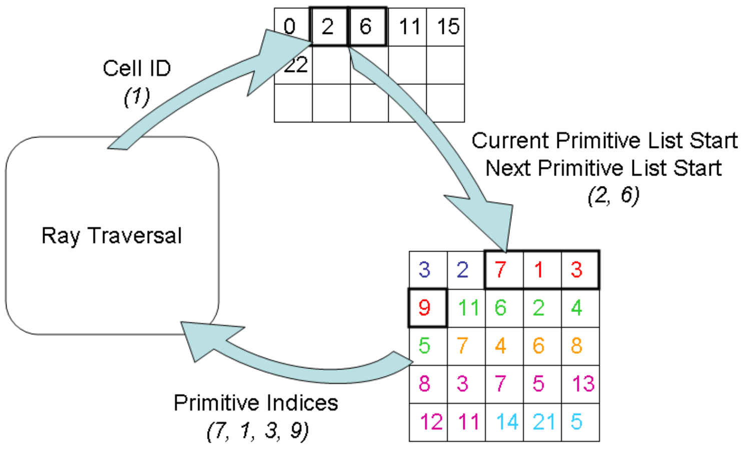


Figure 3.3: The ray traversal procedure uses the current cell ID to obtain its corresponding list of primitives, to be tested for intersection next. Different colors identify separate lists.

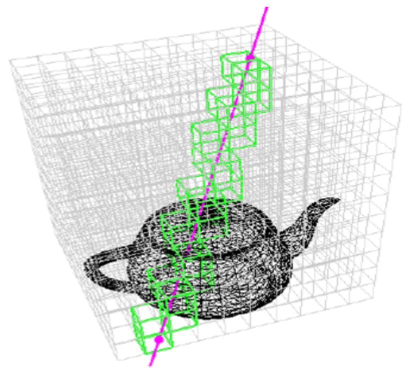
3.2 Accelerating Ray Traversal

The main goal of a ray-traversal procedure is to find the nearest primitive that is intersected by a given ray. Therefore, an acceleration structure must be capable of quickly determining a subset of scene primitives to be tested. Ideally, this subset should be as small as possible. This avoids the need to check for ray-primitive intersection against the entire scene. Additionally, if these primitives can be tested in increasing depth order — that is, along the ray direction — whenever a hit is found the ray-traversal procedure can terminate.

Following this reasoning, the Uniform Grid can be used to traverse the ray through several regions of space, as illustrated in Figure 3.4. Each region, or cell, may contain a number of scene primitives. Whenever a new cell is visited, the procedure checks for ray-triangle intersections and stores the nearest hit. If no hit is found, or if the cell is empty, the traversal continues to the next cell along the ray direction. On the other hand, if the ray intersects a primitive, the algorithm stops the ray traversal and performs the necessary shading computations.

The regular subdivision of a Uniform Grid greatly simplifies the task of determining which cell must be visited next. The classic grid traversal algorithm uses a 3D digital differential analyzer, or 3D-DDA, to step the ray along successive cells [Amanatides and Woo 1987]. The main advantage of this technique is that it only requires a few pre-computed values per ray, and its main loop can be done efficiently on the graphics hardware.

Given a ray origin and direction, the first step is to determine the grid cell where traversal begins. This can be computed efficiently by performing


3.4(a): Cells visited along the ray.

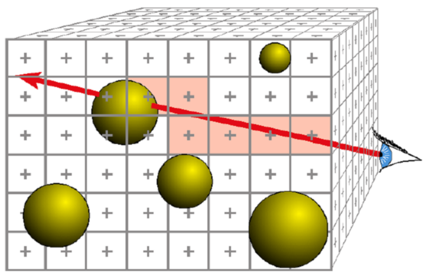

3.4(b): Ray traversal stops as soon as a valid hit is found.

Figure 3.4: Examples of ray traversal using a Uniform Grid.

a simple ray-box intersection test, and evaluating which cell contains the hit position. If no primitive is intersected in this cell, it is necessary to determine which one must be visited afterwards. A direct approach would be to check all neighboring cells for ray intersection. However, it is possible to explore the grid regularity to reduce this decision to only a few multiplications and additions.

During traversal, suppose the algorithm keeps track of the distance from the current position, along the ray, to each boundary of the current cell. In this case, determining which neighboring cell to be visited next corresponds to choosing the smallest value from each distance in X, Y and Z. The 3D-DDA is based on the following observation: cell boundary intervals along each axis are equal. This is a direct consequence of the grid regularity, as can be seen in Figure 3.5.

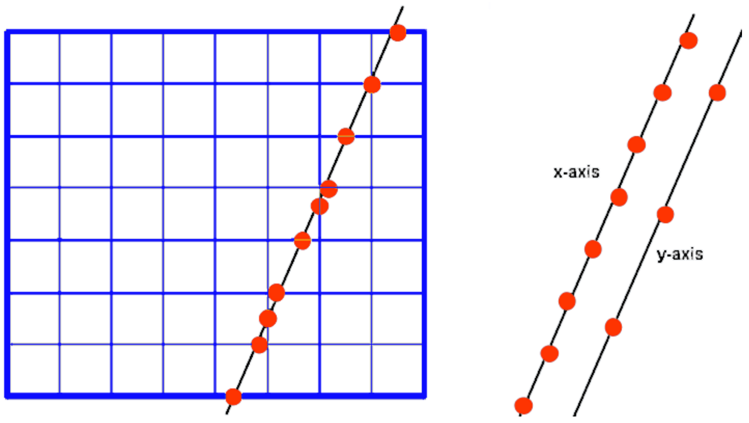
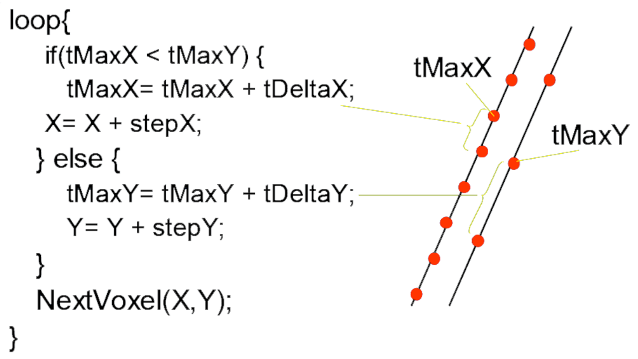


Figure 3.5: The distance between consecutive cell boundaries on each axis is the same.

Considering this insight, updating cell boundary distances during ray traversal becomes a simple task. During ray initialization, the algorithm uses the ray direction to pre-compute the initial boundary distances along each axis, as well as the constant interval that needs to be accumulated at each visited cell. The ray direction also provides the needed information to determine when traversal should stop. That is, the last cell coordinates in X, Y and Z.

The main traversal loop checks, for each cell, which axis boundary is closer. This indicates where traversal should continue. The procedure then updates the current cell coordinates and increments the chosen boundary distance by the regular interval in that axis. Figure 3.6 describes this algorithm in a 2D case, following the previous example shown in Figure 3.5.



```
loop{
   if(tMaxX < tMaxY) {
      tMaxX= tMaxX + tDeltaX;
   X= X + stepX;
  } else {
      tMaxY= tMaxY + tDeltaY;
      Y= Y + stepY;
  }
  NextVoxel(X,Y);
}
```

Figure 3.6: Main traversal loop for the 2D example. Notice that tDeltaX and tDeltaY are constants.

Note that the proposed algorithm consists in a very simple loop. At each step, few multiplications and additions are performed, together with a single control flow instruction. In contrast to other acceleration structures, such as a kd-tree, there is no need to maintain a per-ray traversal stack. These characteristics make the 3D-DDA grid traversal a favorable candidate for implementation inside the GPU.

3.3 Programming the Graphics Hardware

Modern graphics hardware is made of hundreds of specialized processors (as exemplified in Figure 3.7). Each one is capable of performing logic and arithmetic operations, control flow instructions and read/write accesses to caches and a global shared memory. However, since all threads execute in parallel, global inter-processor communication and synchronization is only available at the end of a single program execution (called a *kernel*). State-of-the-art GPUs contain a localized cache for fine-grained thread communication which, usually, is only useful in alleviating memory bandwidth by distributing the same data across neighboring threads.

Each processor is capable of running the same kernel on different data. How computation is parallelized is determined by how the data is structured and how work is distributed among different processors.

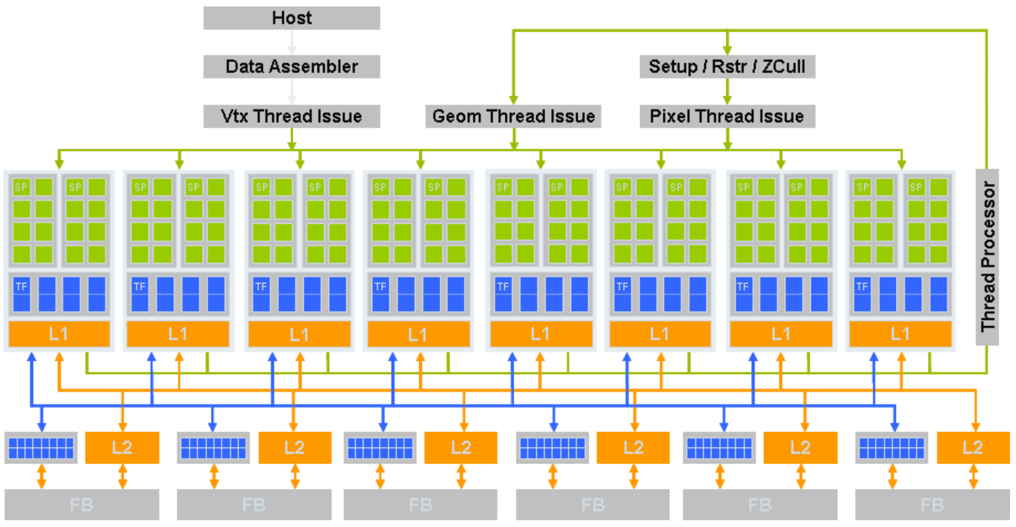


Figure 3.7: This Nvidia GPU is made of 128 general-purpose streaming processors (green boxes) arranged in clusters to share memory and texture functions. Vertex, geometry and pixel processing are load-balanced dynamically during rendering [Nvidia 2006].

The main difficulty in GPU programming is that the hardware is made, first and foremost, for graphics computations. This means that its great parallel processing power is only achieved by making several concessions. The hardware is built around the assumption that several triangles and fragments are being processed independently. Also, there are several mechanisms such as vertex transform and texture caches that assume localized computations typical of traditional graphics applications.

Thousands of threads running in parallel means memory bandwidth is a scarce resource. In addition, memory latency is about a hundred times greater than on the CPU. There is no instruction pre-fetching as well as no local per-processor caches. It is thus necessary to have several threads running in parallel to hide memory latency by performing other threads' computations.

Another restriction is the amount of registers available. A great number of processes can only run efficiently in parallel because context switching is virtually nonexistent. All resources such as registers are pre-allocated for all threads, even inactive ones. This way, to swap processing threads it is only required to change a program counter.

The above factors create a paradox: it is necessary to have thousands of threads to hide memory latency, by doing other computations, but having this number of processes running in parallel consumes a large amount of resources such as memory bandwidth and registers. Therefore, to efficiently solve a problem using the GPU, it must share several similarities to traditional graphics computations. Ideally, the input data should be evenly distributed among different threads, exploiting localized read operations and avoiding concurrent writes to the same memory positions. Also, inter-process dependencies should be kept to a minimum, in order to avoid synchronization barriers and the overhead of several kernel calls.

To use the GPU for general computations, the two best known methods are: shader programming and the CUDA toolkit [Nvidia 2008]. The former uses the traditional OpenGL API together with the GLSL shading language [Rost and Kessenich 2006], or the equivalent Direct3D language called HLSL [Oneppo 2007]. Since it is a more restricted programming model, based upon the traditional graphics pipeline, it is already best suited to fully tapping into the hardware processing power.

On the other hand, the CUDA toolkit provides a new API that is able to remove several hardware constraints (see Figure 3.8). It becomes possible to develop algorithms which rely upon inter-process synchronization and flexible gather/scatter operations. Yet, these benefits are also its main source of criticism. Although more complex algorithms can be adapted to the GPU using CUDA, existing hardware limitations in memory bandwidth, memory access latency and register usage may limit its usability to only small datasets.

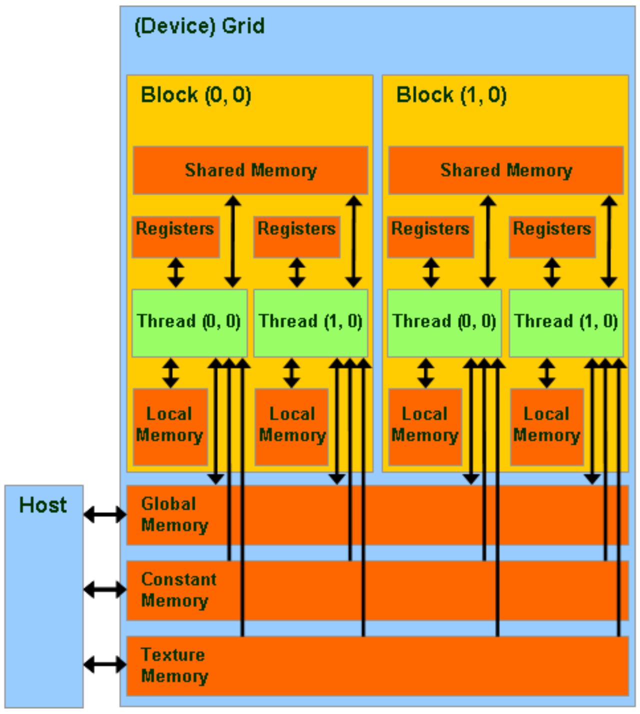


Figure 3.8: In CUDA, a kernel computation is distributed amongst several threads organized in blocks. There are different kinds of memory spaces available, which differ in access latency, available bandwidth, total size and read/write access [Nvidia 2008].

The best possible solution, in our opinion, is to use both techniques where each is better suited. Whenever work could be easily parallelized, with no inter-process dependencies, we have opted for a straightforward GLSL implementation. In contrast, in situations where a complex algorithm, such as sorting, would be required, we have opted for a more elaborate CUDA implementation.

3.4 Ray-Tracing Architecture

Before analyzing implementation-specific details, let us first overview our proposed ray-tracing solution:

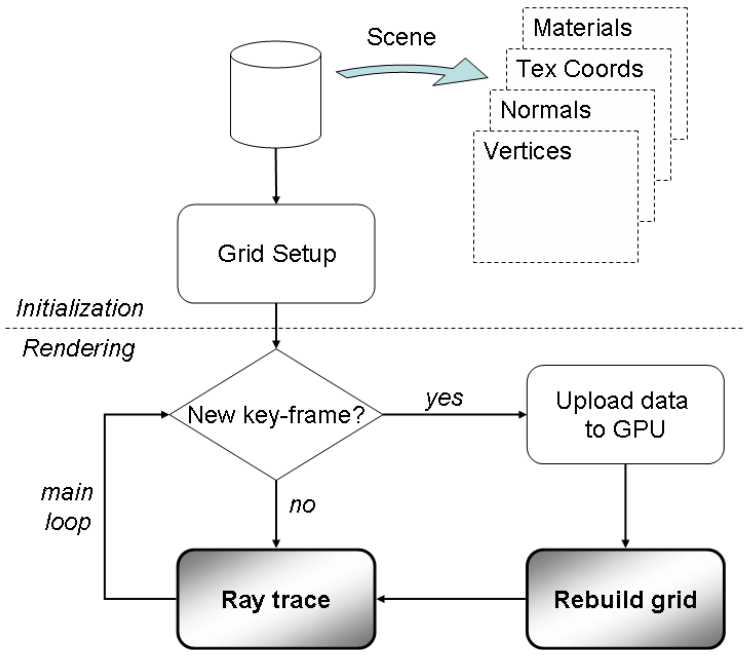


Figure 3.9: General view of the proposed ray-tracing architecture. Highlighted in grey are the two main steps: grid construction and ray tracing.

The first step is to load the scene from the hard-disk. The scene is made of several files that represent geometry and material information. All data is loaded to CPU memory and then uploaded to the GPU as it is needed. This includes vertices, normals, texture coordinates and material descriptions. For an animation made of several key-frames, only data corresponding to the current key-frame is kept inside the GPU.

After loading the scene, the Uniform Grid procedure is configured according to the total number of primitives, the scene bounding box and the desired grid resolution. We also initialize the GLSL and CUDA algorithms, pre-allocating the required memory for rebuilding the entire grid structure.

Following this initialization phase, the main rendering loop begins. At each rendering frame, it is first determined whether a new animation keyframe must be uploaded to the graphics card. In this case, the grid structure must be entirely rebuilt, as illustrated in Figure 3.9.

After that, the ray tracing procedure is performed. This algorithm traverses the ray along the Uniform Grid, while also reading vertex data to compute ray-triangle intersections. Whenever a valid hit is found, the implementation reads the normal, texture coordinates and material information to compute shading results and trace secondary rays. These are used for shadow and reflection effects.

The implementation details for our proposed grid construction procedure is further analyzed in Chapter 4. Afterwards, in Chapter 5, we describe the ray-tracing algorithm used in this work.