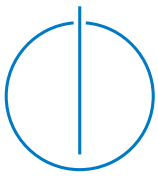
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Real-Time Parallel Streamsurface Computation

Seyedmorteza Mostajabodaveh



DEPARTMENT OF INFORMATICS

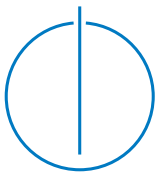
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Real-Time Parallel Streamsurface Computation

Parallele Echtzeitberechnung von Stromflächen

Author:	Seyedmorteza Mostajabodaveh
Supervisor:	Prof.Dr. Rüdiger Westermann
Advisor:	Dr. Andreas Dietrich, Dr. Frank Michel
Submission Date:	15 March 2016



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15 March 2016 Seyedmorteza Mostajabodaveh

Acknowledgments

First, I want to thank all people who has contributed to this work whether directly or indirectly.

Abstract

Streamsurfaces are one of the powerful visualization tools, which are used to gain insight into characteristics and features of flow fields. In practice, streamsurfaces are approximated by triangulating adjacent pairs of integral curves, originating from a seeding line. The generation of integral curves bears quite some similarities to ray tracing algorithms used in physically based renderers. Although, the techniques used in ray tracing may not have good performance in the streamline computation context due to their different computational nature, they can be optimized for streamline computation by introducing some modifications.

In this master thesis, I present my work on accurate streamsurface computation and rendering in real-time, by exploiting the scalability and portability features of parallel architectures in heterogeneous computing, and utilizing concepts from physically based rendering. To improve the efficiency, I use a scheduler to divide the streamsurface computation and rendering tasks on different devices proportional to their computation powers. Additionally, I apply acceleration structures and the concepts of caching to improve the efficiency and utilization of streamsurface generation on modern GPUs and CPUs to achieve real-time results. Furthermore, the possible impact of applying ray-packing and ray-sorting to the streamline computation is investigated.

Contents

1. Introduction

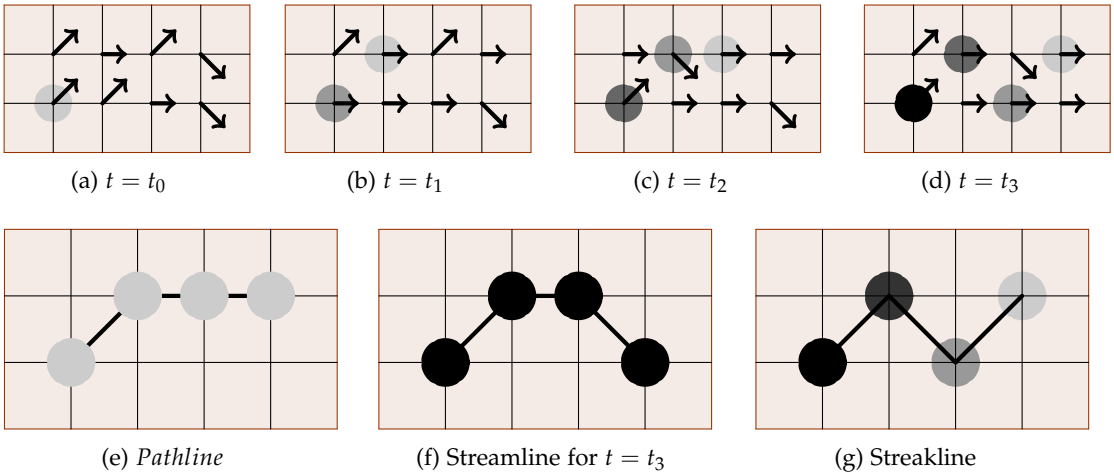
The successful interpretation of a physical process is significantly improved if it is visually observable. Some natural phenomenon like a fluid flowing in a channel or around a solid obstacle are observable by the human eye, but some others like the ones involving transparent materials or invisible flows, e.g., magnetic fields are not observable by the human eyes. Just observing a phenomenon is not enough to interpret it. For example, some fluids form very complicated patterns, which human intuition fails to imagine. Although in such cases the patterns in the flow fields can be analyzed with mathematical methods, there are complicated flows containing patterns which are impossible to understand without getting advice from visual images of the flow. For instance, some flows are very sensitive to small changes in geometry or boundary conditions. For example a slight increase in noise or roughness in laminar flow can make it turbulent.

1.1. Flow Visualization

Flow fields can be of two different types: *steady* flow fields which are independent of time, and *unsteady* flow fields, which are dependent on time. In the later, patterns succeed one another. This makes interpretation even harder. In order to recognize such patterns, one must provide some methods to make the flow visually visible. These techniques are called *flow visualization*. Flow patterns can be visualized in many different ways, e.g., injecting dye or smoke in a flow field, or photographing particles seeded in flow field with long time exposure.

The life cycle of a numerical flow simulation consists of three phases: grid generation, flow calculation, and visualization. Depending on the visualization method, the cycle results in one or more images showing flow patterns, which makes the interpretation of the flow field's physical features possible. Basic visualization methods for generating images of flow patterns are pathlines, streamlines, and streaklines[**Johnson:2004:VH:993936**]. The definition of these tools are as follows:

- a **pathline** shows the trajectory of a single massless particle released from a fixed location, called the *seed location* (Figure 1.1e).



(a) $t = t_0$ (b) $t = t_1$ (c) $t = t_2$ (d) $t = t_3$

(e) *Pathline* (f) Streamline for $t = t_3$ (g) Streakline

Figure 1.1.: **A comparison between pathline, streamline and streakline** is shown in this example. A pathline is trajectory of a massless particle in an unsteady flow, a streamline is a trajectory of a massless particle in a steady (frozen) flow, and a streakline shows trace of dye that is released into an unsteady flow at a fixed position.

- a **Streamline** is a field line tangent to the velocity field at an instant of time in a steady flow (Figure 1.1f).
- a **Streakline** is a line joining the positions of all particles, at an instant of time, that have been previously released from a seed location over time in an unsteady flow (Figure 1.1g).

The 3d generalized forms of streamlines and streaklines are called **streamsurface** and **streaksurface**. These surfaces are powerful visualization tools, which are used to gain insight into characteristics and features of the 3d flow. Streamsurfaces are mainly used for steady flows and streaksurfaces are used for unsteady flows. In practice, these surfaces are approximated by triangulating adjacent pairs of streamlines or streaklines, which originate from a seeding curve (see section 2.1).

Figure 1.2 shows an example of using streamlines. The aircraft directs its vectoring nozzles downward to generate a lift force, so it can hover in the air. The red streamlines visualize the flow clearly. As the exhaust is exiting downward with a high velocity. Figure 1.3 shows another example. It presents how streamsurface and streaksurface are visualized in the so-called von Karman dataset. The von Karman dataset is a an unsteady flow of fluid which goes around an obstacle. A vortex is made behind the

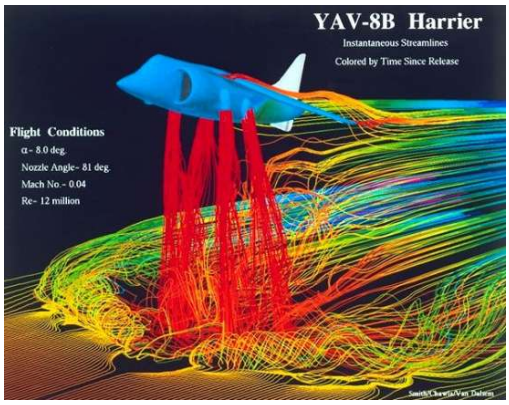


Figure 1.2.: **Visualizing the flow with streamlines**. The image shows how a Harrier aircraft is pushing itself against the ground by directing its vectoring nozzles downward to generate a propulsive force while it is hovering. Red streamlines visualize the high-velocity exhaust exiting downward to keep the aircraft airborne.

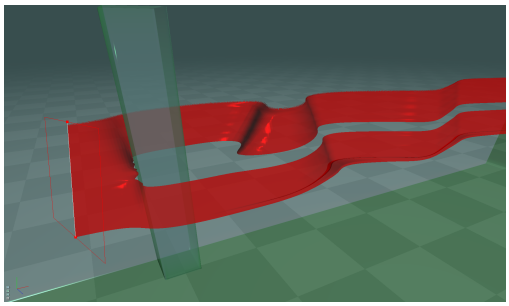


Figure 1.3.: **Streamsurface representation** in the von karman flow field. Computed and visualized a streamsurface in a frozen flow (at a specific time step).

obstacle which start moving inside fluid.

Nowadays, scientists are performing complex 3d unsteady and steady flow simulations which results in simulation data organized in grids (see section 2.1). Table 1.1 shows the information about five different grids which have been used in this master thesis. Each simulation is made of two parts: one or more *grid files* which contains information about the mesh of the grid, and one or more *solution files* which contains the momentum, pressure or any other quantity measured or computed in the flow calculation. The size of the grid depends on the number of grid points, and the size of the solution file depends on the size and the number of quantities measured or computed during simulation. In unsteady flows, a solution file is stored for each time

Grid	Points	Grid size per time step	Solution size per time step
Wind Tunnel	368,606	176 MB	117 MB
2d Karman	1,044,936	75.7 MB	43.7 MB
Karman	16,203,918	1.59 GB	1.61 GB
Thetrahedradecal (v2)	37,953,007	4.62 GB	2.76 GB
Thetrahedradecal (v3)	29,996,940	6.61 GB	1.49 GB

Table 1.1.: Simulations used for steady/unsteady flow visualization

step. In some cases, the grid changes over time, therefore each time step contains one grid file, also. Common unsteady flows are usually made of thousands of time steps. For example, the 3d von Karman simulation is made of $1,000$ time steps. As, the grid is not changing during the simulation, the whole dataset consists of one grid file and 1000 solution files. Therefore, the unsteady flow requires around 1.62 TB on disk (1.59 GB$+1000 * 1.61$ GB ≈ 1.61 TB).

1.2. Heterogeneous Computing

Real-time flow visualization demands more memory and computation power for bigger datasets with higher complexities. Nowadays, computers benefit from two powerful computation units: the *central processing unit* (CPU) and the *graphics processing unit* (GPU). Each of these units has its own dedicated memory. This is an opportunity to utilize concurrency and improve performance by mapping tasks to the available processing devices on the system. In the last few years, there were many efforts to bring *heterogeneous computing* to a single device offering both, GPU and CPU features. Hardware vendors have released different products, but as design and architecture of these computation units are different, classical programming models do not work on both of them very well.

The main contribution of this master thesis is to compute and visualize high-quality streamsurfaces in real-time for datasets that fits into the memory of all computation devices. Heterogeneous computing is applied in this context to map the computation and rendering tasks to appropriate devices in parallel to achieve the highest possible performance. Additionally, high-quality integration methods, i.e., adaptive Runge-Kutta, and several other sophisticated techniques are utilized for fast and accurate streamsurface computation. Furthermore, generation of streamlines bears similarities to ray tracing algorithms used in physically based renderers. In the last decade, tremendous progress has been made in real-time ray tracing, with performance improvements of several

orders of magnitude. Another objective of this thesis is to investigate if techniques from the real-time ray tracing field, e.g., acceleration structures, or ray sorting algorithms can also be applied to the area of streamline/surface generation.

1.3. Use Case: Real-Time Prototype Visualization

Aerospace and automotive researchers use wind tunnels to study the effect of air moving past solid objects. They usually put a prototype model instrumented with force, pressure, and other sensors into a wind tunnel. Then, they measure the aerodynamic forces at specific points in the tunnel at defined time intervals. This process is very critical for designers of cars and airplanes. Researchers use streamlines, streaklines, streamsurfaces, and streaksurfaces to visualize the patterns in the flow at different points. They carefully interpret these images to make critical changes or improvements to their designs. Today, *computation fluid dynamics* (CFD) modeling and simulation on computers, are replacing the wind tunnel tests.

In this part, an intuitive use case for using streamlines, streaklines, etc. is a tool features design, simulate, and visualize flow around a prototype model in wind tunnel easily and fast.The designer can use computer-aided methods to design a 3d model of the prototype, simulate the flow around the model in the wind tunnel, and visualize the flow and interpret it precisely. He loads the flow at a specific time step into the tool, and wear *virtual reality* (VR) devices. He is able to use the VR device to enter the virtual world, and interact with it. As an example, he can use his hands to place the starting point of the streamlines or streamsurfaces in the virtual world. He is also able to move around the model and look very closely at computed streamlines and streamsurfaces, check patterns of the flow, and verify if they are as he expected. The designer can iterate this procedure until he can achieve the appropriate design of the model.

The last phase must be realtime as the display should response to the designer's interactions immediately. The output image displayed bears to be of high quality (without any juddering). In addition to the visual quality of the final output, the streamlines/streamsurfaces should be of high accuracy. To achieve these goals, very high computation power is required. Heterogeneous computing can be utilized in systems with multiple CPUs and GPUs to perform computations and renderings concurrently.

Figure 1.4 shows the Immersive Streamlines project done at Fraunhofer IGD. In this demo, the user wears a VR device consisting of a *head mounted display* (HMD) and a hand tracker. There is also an infrared camera which tracks the user position. He is able to move around the virtual environment and look closely at the objects and streamlines. Also, he is able to move the starting point of the streamlines and explore

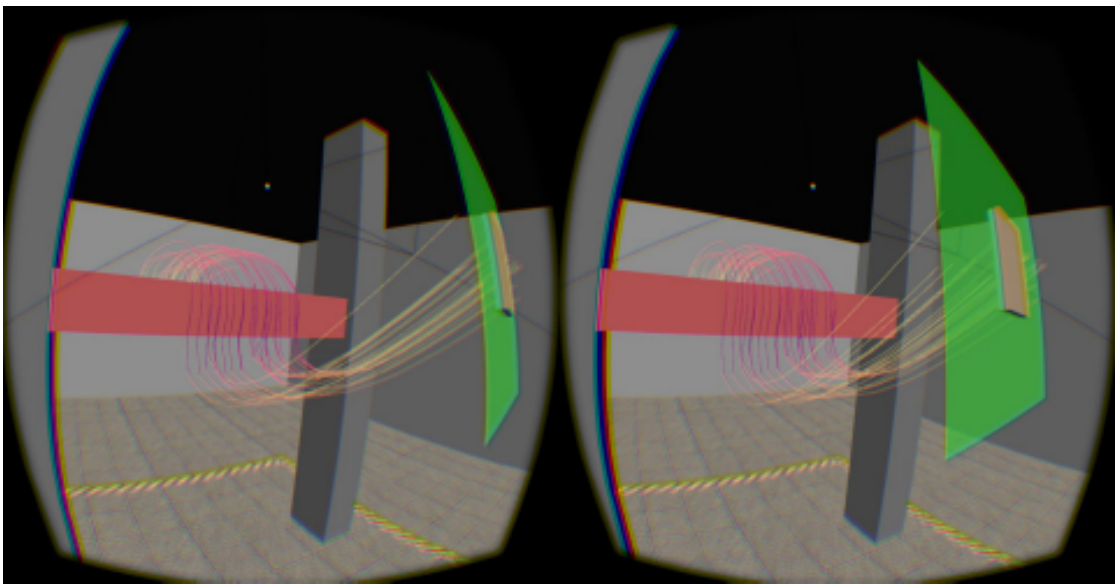


Figure 1.4.: **Immersive Streamlines** is a prototype demo done at Fraunhofer IGD. The user is able to move inside the virtual world, interact with it, and look closely at the flow visualization results. The image shows a stereo pair, which is displayed on Oculus Rift HMD.

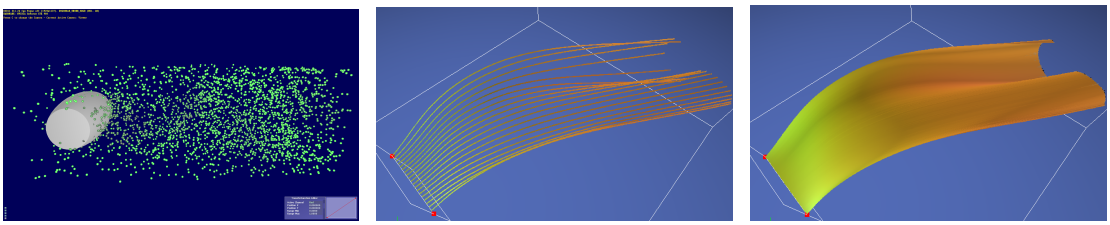
different patterns of the flow. This method of interaction helps scientists and designers to get a good understanding of the flow features.

2. Background

2.1. Stream Tracing

In the previous chapter, visualization techniques used to help the understanding of patterns in flow fields was discussed. Particle/streamline/streamsurface computation in general are called *stream tracing* techniques. They have been discussed informally in the previous chapter. However, they have mathematical definitions [**6378974**].

Assume a stationary flow field ($\mathcal{V}(s)$) defined in $\Omega \subset \mathbb{R}^3$. A *streamline* $\mathcal{S}_{x_0}$ is a solution of the ordinary differential equation $\dot{\mathcal{S}}_{x_0}(t) = \vec{\mathcal{V}}(t)$ with initial condition $\mathcal{S}_{x_0}(0) = x_0$. In other words, $\mathcal{S}$ is a curve starting from x_0 and tangent to the flow field at every point. The point x_0 is called *seeding point*. As discussed before, a streamline can also be defined as a trajectory of a massless particle advected through steady flow field from point x_0. If $\mathcal{C}(s)$ is a curve defined in Ω, *streamsurface* ξ can be mathematically represented as $\xi(s;t) := \xi_{\mathcal{C}(s)}(t)$, where s is a parameter used for sampling seeding points on the seeding curve and t is used to advance front the integral curves, i.e., the union of streamlines starting on a curve called *seeding curve* $\mathcal{C}(s)$ is a streamsurface.



(a) Particles (b) Streamlines (c) Streamsurface

Figure 2.1.: **Examples of stream tracing techniques** examples. Particle tracing, streamline tracing, and streamsurface tracing is shown in this figure.

In practice, streamsurfaces are approximated by *triangulating* N streamlines starting from points sampled at equal distance along the seeding curve. Usually this process involves adding new points/streamlines in-between, which is called *refinement*. Streamlines are computed by tracing particles from the seeding points with a specific step size. Tracing a particle in a flow field is called *particle tracing*. Practically, it is done by

advancing a particle through a flow field in small steps. Figure 2.1 shows examples of these stream tracing techniques.

2.1.1. Data Representation

Stream tracing visualization techniques take a flow field as input. Current computers used for acquiring, analyzing, and representing flow fields are digital computers, the data stored/retrieved in/from them are *discrete*. One problem of using discrete data is that the value between two data entries is not available but in many flow visualization techniques, determining data values at arbitrary positions is important. The solution is to *interpolate* between two neighboring values.

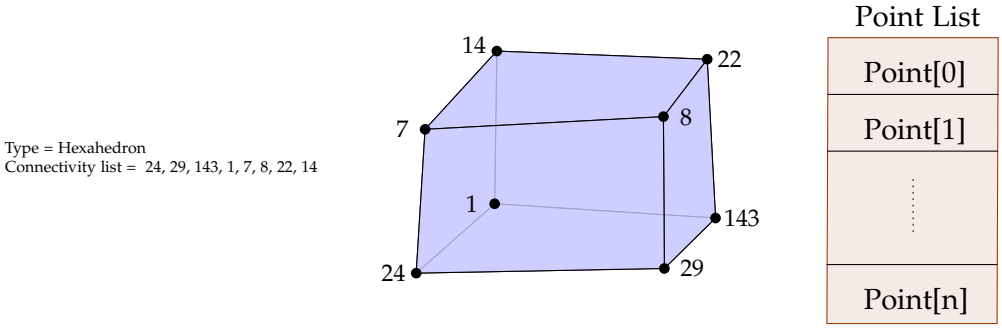
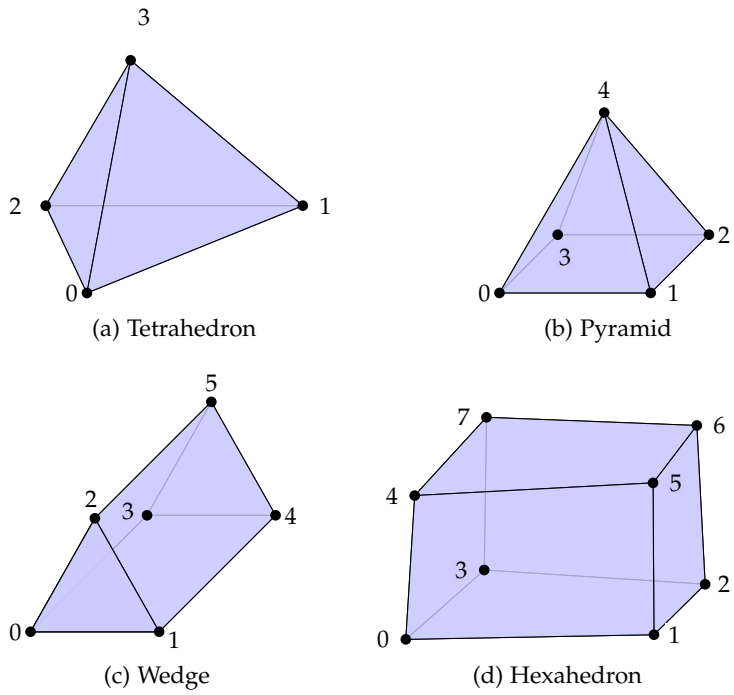


Figure 2.2.: **Examples of stream tracing techniques** examples. Particle tracing, streamline tracing, and streamsurface tracing is shown in this figure.

Flow field dataset consists of: *organization structure* and associated data *attributes*. The organization structure consists of *topology* which does not change under geometric transformation, and *geometry* which is an instance of the topology specified in 3D space. E.g. a polygon is a triangle, defines the topology of shape but when the coordinates of the corners are defined, the geometry is specified. Attributes are additional information about the topology or geometry e.g. pressure or velocity at a point. Flow field dataset is defined with one or more *cells* specifying topology of dataset. For each cell, in addition to cell type, an ordered list of points (known as *connectivity list*) is provided to define the geometry of dataset. Each cell or point can accompany with additional information like pressure or velocity inside the cell or at specific point. Figure 2.2 shows how a cell is defined.

The cells can be 0/1/2/3-dimensions. The dataset used in this master thesis, has only 3d cells therefore the ones with less dimensions is not described here. There are an infinite variety of 3d cell types. All these cell types can be subdivided into four



(a) Tetrahedron (b) Pyramid

(c) Wedge (d) Hexahedron

Figure 2.3.: **Primary cell type shapes**.

primary cell types (figure 2.3):

- **Tetrahedron**, defined by four non-planar points. It has six edges and four triangular faces (Figure 2.3a).
- **Pyramid**, a shape with five points which four of them define a quad plane and the fifth non-planar apex point. It consists of five points, eight edges, forming one quad and four triangles (Figure 2.3b).
- **Wedge**, consists of three quads and two triangular faces. It is defined by six points and nine edges. It has a convex shape that its faces and edges do not intersect each other (Figure 2.3c).
- **Hexahedron**, consists of eight points, six quads as faces, and twelve edges. The edges and faces should not intersect (Figure 2.3d).

Figure 2.3 presents the shape of different cells. As these cells are linear, linear interpolation can be applied for extracting the information in an arbitrary point. The flow field datasets are organized in a structure of above cells called *grid*. It is composed

of a list of points, and cells. The grid made of 3d cells can be of two types: *structured grid*, and *unstructured grid*.

The structured grid has a regular topology and irregular geometry. It means to specify a cell into a grid, specifying a 3d vector (n_x, n_y, n_z) is enough. This type of grid is mostly made of one type of cell. For geometry of the grid is represented by going through the cells and maintaining each cell point coordinates. This type of grid is used in most of fluid flow simulations. The unstructured grid is the most general form of flow dataset. Both topology and geometry are irregular. It can be made of any cell type. This type of grid requires more memory and computation than structured one. Therefore, scientist use the unstructured grid when structured one cannot be used. In this master thesis unstructured grids are used for storing the flow fields.

Datasets used in this master thesis are of two formats: *CFD General Notation System* (CGNS), and *Open-source Field Operation and Manipulation* (OpenFOAM). There are some conventions to store/retrieve the cells, points, and attributes data. C++ libraries called CGNSlib and VTK can be used for retrieving these information. OpenFOAM files can be made of different domains, and CGNS files can be made of multiple zones but datasets used here are single domain/zone flow fields.

2.1.2. Advancing Particles

As discussed before, streamline and streamsurface computation is based on advecting a particle through the flow field. It can thought as moving continuously a massless object along the velocity at its position for a small time step. As, velocity $\vec{\mathcal{V}} = dx/dt$, the displacement of the point is defined as $dx = \vec{\mathcal{V}}dt$. The displacement path of the object can be expressed as $\vec{x}(t) = \int_t \vec{V}dt$. In a steady flow field, this equation represents streamlines. Although, In most cases it cannot be solved analytically, solution can be approximated using *numerical integration* techniques. *Euler* method is its simplest form:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\mathcal{V}}_i\Delta_t$$

where $\vec{x}_{i+1}$ is the next position of the object, $\vec{x}_i$ is its current position, $\vec{\mathcal{V}}_i$ is the it's velocity in current position, and Δ_t is the step size. Smaller step sizes result in more accurate numerical integration. Euler method's error is of order $\mathcal{O}(\Delta_t^2)$. There is another class of numerical integration methods which is called *Runge-Kutta*. For example, Runge-Kutta of order 2 formula is as follows:

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta_t}{t}(\vec{\mathcal{V}}_i + \vec{\mathcal{V}}_{i+1})$$

The error of this method is of order $\mathcal{O}(\Delta_t^3)$. As, the error order is smaller than Euler's method, larger steps can be used. Although it increases integration speed, the formula

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$	$\ddots$			
c_s	a_{s1}	a_{s2}	$\dots$	$a_{s,s-1}$	
	b_1	b_2	$\dots$	b_{s-1}	b_s

Table 2.1.: Butcher tableau general representation

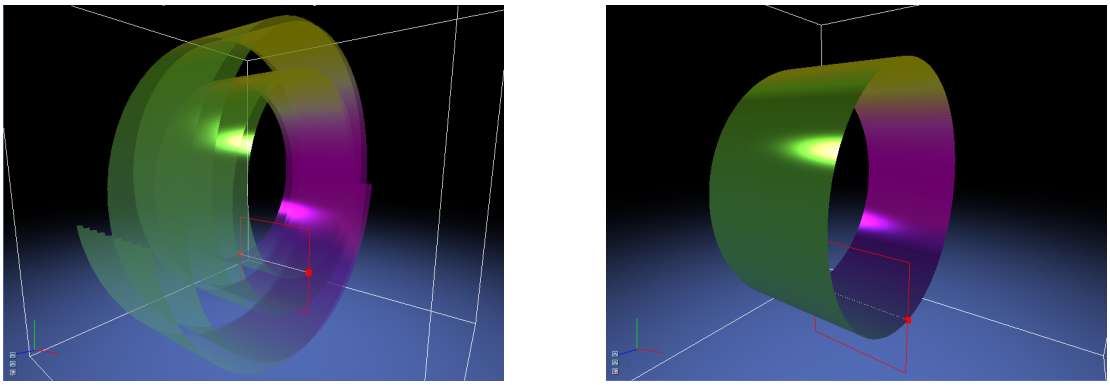
contains one additional velocity evaluation at next point computed with Euler method which is costly.

There are two different runge-kutta methods: *implicit* and *explicit*. The explicit runge-kutta methods can be described using a table called *Butcher tableau*. Table 2.1 shows a general representation of this table. The $a_{i,j}$ is called runge-kutta *coefficients matrix*, c_i are called *nodes*, and b_i are called *weights*. To specify a particular method, the number of steps (s), coefficients matrix, nodes, and weights should be determined. The integration is done using the following formula:

$$x_{n+1} = x_n + \sum_{i=0}^{s} b_i * q_i$$

where,
$\vec{q}_1 = \vec{\mathcal{V}}(x_i)$
$\vec{q}_2 = \vec{\mathcal{V}}(x_i + \Delta_t * a_{21} * q_1)$
$\vec{q}_3 = \vec{\mathcal{V}}(x_i + \Delta_t * (a_{31} * q_1 + a_{32} * q_2))$
$\vdots$
$\vec{q}_s = \vec{\mathcal{V}}(x_i + \Delta_t * (a_{s1} * q_1 + a_{s2} * q_2 + \cdots + a_{s,s-1} * q_{s-1}))$

One of the widely used explicit runge-kutta methods is called *adaptive runge-kutta*. The local error of a single step of these integration methods can be calculated. It is a convenient criterion to manipulate the step size. In other words, the step size should be increased if the local error is smaller than specific threshold and the it should be decreased if the local error is larger than a predefined value. Cash-Karp is one of adaptive runge-kutta methods with error of order $\mathcal{O}(\Delta_t^6)$. Butcher tableau for Cash-karp coefficients can be found in table 2.2. There are two sets of weights available for this method. The first row has accuracy of order $\mathcal{O}(\Delta_t^5)$ and second row is of order $\mathcal{O}(\Delta_t^4)$. The difference between two solutions is the local error of this method.



(a) Euler (b) Cash-Karp

Figure 2.4.: **Comparing the euler and cash-karp integration methods** in a perfect rotation around central point flow field. The step sizes for both integration methods are the same. The result stream surface should be a top and bottom opened cylinder as it is in 2.4b while the euler method result is has a spiral shape.

2.1.3. Cell Locating

To advect a particle through the dataset, the velocity vector in sampling point should be looked up. The lookup will be costly if all the cells of the grid are tested against the sampling point. It is more efficient to test the point against the bounding boxes of the cells before performing point-in-cell test. This test calculates the barycentric coordinates of the point in cell parametric coordinates. If they are between zero and one, the point is inside the cell otherwise the point is outside of the cell. Furthermore, the barycentric coordinates can be used for interpolating the attributes inside the cell.

Table 2.3 contains the interpolation functions for calculating the barycentric coordinates for a point inside/outside of a cell. For each cell type, a parameter coordinate system is specified. To calculate the barycentric coordinates for a point, it should be transformed to (r, s, t) coordinate system. These values should be replaced in formula provided in barycentric coordinates column of table. Point is residing into cell if all of them are between 0 and 1. Interpolated attribute can be computed by using the barycentric coordinates (W_i) using $\sum_{i=1}^{N} W_i * Attrib_i$ formula.

While point-inside-AABB is more efficient than point-inside-cell but checking all the cells toward the sampling point is costly. Therefore, a special kind of *spatial structure* is required to narrow down the number of candidate cells. The goal for any cell location scheme is to keep the number of candidates cells small. There are four well-known

0						
1/5	1/5					
3/10	3/40	9/40				
3/5	3/10	-9/10	6/5			
1	-11/54	5/2	-70/27	35/27		
7/8	1631/55296	175/512	575/13824	44275/110592	253/4096	
	37/378	0	250/621	125/594	0	512/1771
	2825/27648	0	18575/48384	13525/55296	277/14336	1/4

Table 2.2.: Cash-karp butcher tableau. It is used to represent the runge-kutta coefficient matrix, nodes, and weights to integrate a point.

acceleration structures used in ray-tracing techniques. Although, chapter 3 will describe these acceleration structures in detail, here a general description of each is presented:

- **Grid** divides the region of space into axis-aligned equal-sized voxels. Each voxel stores the indices of the cells which their bounding boxes are overlapping it. Given a sample point, the voxel containing the cell can be found with $\mathcal{O}(1)$. To locate the target cell, sample point should be checked inside bounding boxes and cells stored in the voxel. This process also is done in $\mathcal{O}(n)$. The primitive may overlap with more than one voxel, in this case it should be stored in more than one voxel. In this case, the size of the grid on the memory and the number of candidate primitives to be tested for intersection will increase (Figure 2.5a).

- **Bounding Volume Hierarchy (BVH)** is a hierarchy of primitives generated by partitioning them into disjoint sets. This approach is based on *primitive subdivision*. The goal is to generate a hierarchy which the primitives are stored in the leaf nodes, and interior nodes contain the bounding box of beneath primitives. As, the subdivision is done based on primitives, each primitive appears once in hierarchy. Therefore, the number of nodes generated a BVH which stores one primitive per leaf node is $2n - 1$ (n leaf nodes + $n - 1$ interior nodes). In ray tracing, the BVH is the most efficient acceleration structure because generating it is as efficient as a grid and has a higher performance than other ones (Figure 2.5c).

- **KD-Tree** is an acceleration structure which is built based on *space subdivision* approach. Basically it is a *binary space partitioning (BSP)* tree which each of its interior nodes has two children. The leaf nodes contain the primitives overlapping with them. In this approach, a primitive may overlap more than one leaf node and will be present in more than one leaf node. While, It will increase the memory

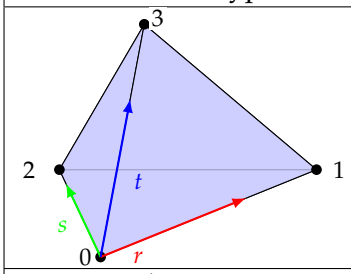
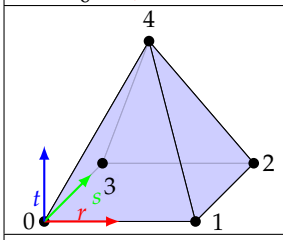
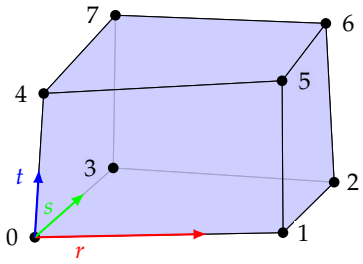
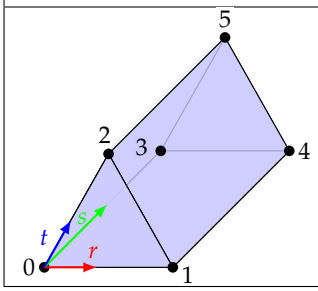
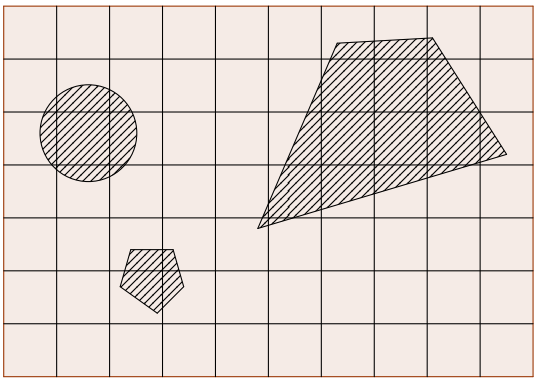
Cell Type	Barycentric Coordinates
	$W_0 = 1 - r - s - t$ $W_1 = r$ $W_2 = s$ $W_3 = t$
	$W_0 = (1-r)(1-s)(1-t)$ $W_1 = r(1-s)(1-t)$ $W_2 = rs(1-t)$ $W_4 = (1-r)s(1-t)$ $W_4 = t$
	$W_0 = (1-r)(1-s)(1-t)$ $W_1 = r(1-s)(1-t)$ $W_2 = rs(1-t)$ $W_3 = (1-r)s(1-t)$ $W_4 = (1-r)(1-s)t$ $W_5 = r(1-s)t$ $W_6 = rst$ $W_7 = (1-r)st$
	$W_0 = (1-r-s)(1-t)$ $W_1 = r(1-t)$ $W_2 = s(1-t)$ $W_3 = (1-r-s)t$ $W_4 = rt$ $W_5 = st$

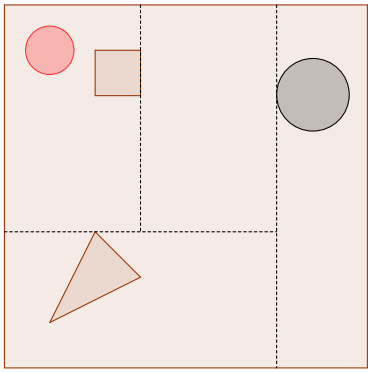
Table 2.3.: **Parameter coordinate system and interpolation function** used for different cell types.

requirement of the tree and number of primitives to be tested for testing the intersection, no stack is required for traversing the tree (Figure 2.5b).

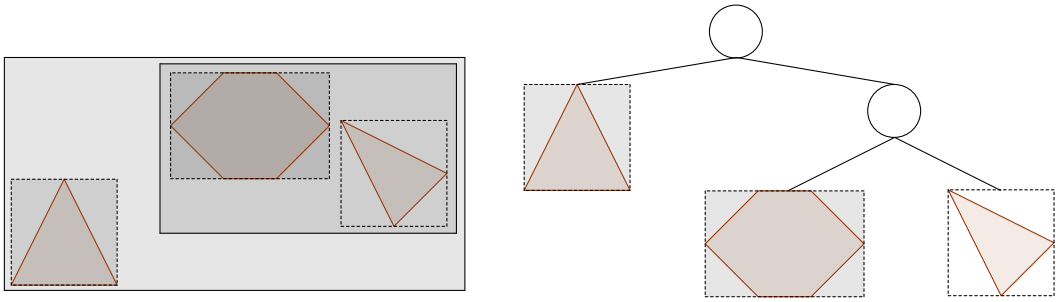
- **Bounding Interval Hierarchy (BIH)** is a combination of BVH and kd-tree. The idea behind this acceleration structure is to partition the primitives using 2 planes



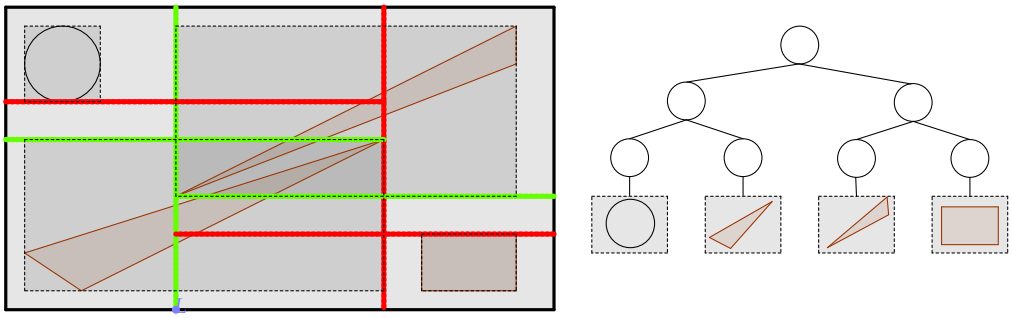
(a) Grid



(b) Kd-tree



(c) Bounding Volume Hierarachy



(d) Bounding Interval Hierarachy

Figure 2.5.: **Different Acceleration Structures** used for cell locating faster. Also, they are used frequently in ray tracing algorithms.

instead of 1 (kd-tree) and 6 (AABB in BVH). The construction and storage of this structure is very similar to BVH trees but the traversing strategy resembles

Kd-tree. Furthermore, it is using a space partitioning approach which make it comparable to kd-tree. Additionally, Like BVH tree each primitive is only stored in one leaf node, so it will make this structure memory efficient. (Figure 2.5d)

2.2. Heterogeneous Computing

Consider the code listing 2.1 written in C++ language and listing 2.2 written in CUDA. The first one runs on CPUs and the latter executes on GPUs but neither the CPU code can be run on GPUs nor vice versa. Therefore, In each case the developer should ignore the other devices and specialize the code for a specific processor. This needs too much efforts for developer.

Heterogeneous computing (HC) means using more than one kind of processor in a system. To utilize all these processors, the developer should provide different versions of code, specific to each processor. *Open Computing Language* (OpenCL) gives the developer the possibility to write one kernel and execute it on both CPU and GPU. While the same chunk of code can be run on all computation units, it is possible to execute two chunks of code on two different devices, concurrently. The main goal of HC is to utilize the bests of CPUs and GPUs to improve the overall performance and power consumption. For instance, vector processors like the one used in GPUs benefit from thousands of individual cores which can operate simultaneously, but sometimes scalar approach of CPUs can solve some problems faster. That's the situation where HC can help by executing scalar algorithms on CPUs while the vectorized ones are running on GPUs, concurrently.

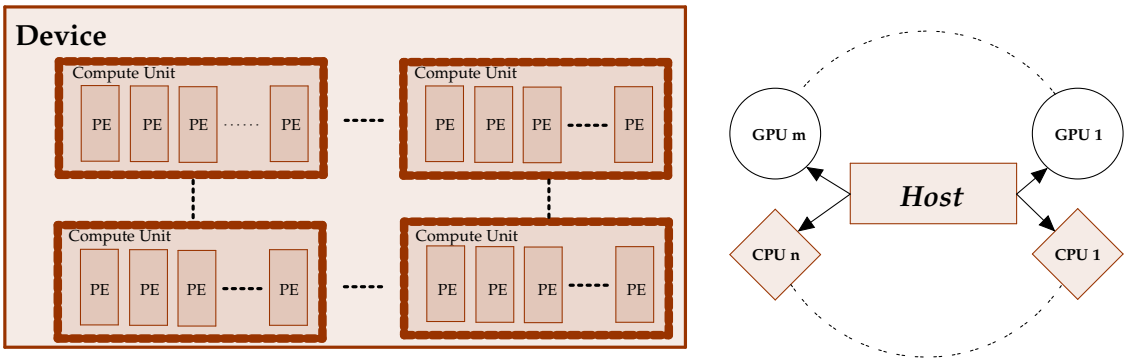
```
void add_vectors(const int& n_elements, const std::vector<int>& a, const std::vector<int>& b,
        std::vector<int>& c){
  for (int i=0; i < n_elements; i++)
    c[i] = a[i] + b[i];
}
```

Listing 2.1: Adding two vectors CPU code in C++

```
__global__ void add_kernel(int n_elements, int *a, int *b,int *c){

  int threadId = blockIdx.x *blockDim.x + threadIdx.x;
  if(threadId >= n_elements) return;

  c[threadId] = a[threadId] + b[threadId];
}
```

Listing 2.2: Adding two vectors GPU code in CUDA

OpenCL was initially introduced by Apple Incorporation. Afterwards, the Khronos Group (the group developing OpenGL, OpenGL ES, and so on) continues developing the OpenCL. Currently, the latest version released by Khronos is OpenCL 2.1 which AMD and Intel has driver implementations for it. Unfortunately, Nvidia is not supporting the newest version of OpenCL. Nvidia has started supporting OpenCL 1.2 in its recent graphics drivers. As, this master thesis should be compiled and run platform independent, OpenCL 1.1 is used in its implementation and some of the brilliant features of new OpenCL versions are not used. Two of them which could affect the performance of the streamline/surface computation are **Shared Virtual Memory (SVM)** which can remove expensive data transfers for complex pointer-based structures like trees and linked lists by sharing them between different devices, and **Dynamic Parallelism** which allows launching kernels in a another kernel without host interference which can remove bottlenecks in the code.



(a) Device (b) Host-Device (CPU/GPU)

Figure 2.6.: **OpenCL platform model** which consists of a host coordinating the main direction of the program while it can send kernel execution commands to other computation devices. Generally, a device is composed of compute units. Furthermore, Compute units consist of processing elements.

The OpenCL platform consists of a host processor which controls the execution of main program. It can direct other processors to perform works (Figure 2.6). Each device processor is made of *compute units*. Compute units consists of multiple *processing elements*. Vendors map this device architecture to the physical hardware to support the OpenCL platform model. As an example, Nvidia GTX 980 has 16 Maxwell streaming multiprocessor called SMM (compute units). Each compute unit has 128 CUDA core processing blocks (processing elements). Each CUDA core is able to execute one instruction at time. Therefore, GTX 980 can totally execute 2048 instruction at each clock.

To execute OpenCL kernels on one/more computation devices, a few steps are necessary which are listed as follows:

- **Detecting *platforms* and *devices***: Platform groups the opencl devices which can interact with each other. It is usually vendor-specific e.g. Nvidia platform has Nvidia gpus inside. As the initial step, the platforms and devices supporting the OpenCL should be detected. OpenCL API provides some querying functions to get different information about detected platform and their devices. Listing 2.3 show how to query for platforms. In this small code chunk, all available platforms are detected and stored in an array. All devices associated with $platform[0]$ are detected and stored in another array.

```
cl_int error; //!< Stores the error status

// store information about platforms and devices
cl_uint num_platforms = 0, num_devices = 0;
std::vector<cl_platform_id> platforms;
std::vector<cl_device_id> devices;

// retreiving platforms
error = clGetPlatformIDs(0, NULL, &num_platforms);
platforms.resize(num_platforms);
error = clGetPlatformIDs(num_platforms, platforms.data(), nullptr);

// retreiving the devices of platform[0]
error = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, nullptr, &num_devices);
devices.resize(num_devices);
error =
    clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, num_devices, devices.data(), nullptr);
```

Listing 2.3: Detecting OpenCL platforms and devices

- ***Context* creation**: Context is handling the host-device interaction, manage memory objects for available devices, and keep track of programs and kernels created or executing for each detected device. After detecting the devices, a context should be created for them. Devices associated with a context and its properties are determined in this step. The memory and program created in one context can be used on all devices used to create the context. In Listing 2.4, a context is created for all devices of the first platform. Here, no specific context property is passed.

```
cl_context context =
    clCreateContext(nullptr, num_devices, devices.data(), nullptr, nullptr, &error);
```

Listing 2.4: create opencl context shared between the devices passed as argument

- **Creating *command-queues* per device**: Command-queue is the mechanism used by host to send its action requests. Each device needs a command-queue which keep track of incoming action requests. While, the commands usually needed to be run in-order, it is possible to set the command execution out-of-order. In listing 2.5, for each device a command-queue is created. The commands for these devices will be run in-order.

```
std::vector<cl_command_queue> cmd_queues(num_devices);
for(cl_uint d = 0; d < num_devices; d++)
        cmd_queues[d] = clCreateCommandQueue(context, devices[d], 0, &error);
```

Listing 2.5: create opencl command-queue per device

- **Creating and filling memory objects** The data which OpenCL kernels operate on needs to be allocated and copied to specific OpenCL buffers. To create a buffer, context containing it and its size should be determined. The buffer created on a context is accessible by all of associated devices. The copy from a host pointer to a buffer can be done using OpenCL API function. The API is getting the command-queue for a device as input, because it is copying the data directly to it. It is possible to set the copying operation as blocking or non-blocking. In listing 2.6, three different buffers are created which two of them are just read in the kernel, because the data for those two will be provided by host. The third one is a buffer which the kernel will only write inside as output. These buffers are being prepared for adding two vectors of arbitrary size into another buffer.

```
std::vector<int> A(VEC_LEN), B(VEC_LEN), C(VEC_LEN);
size_t buf_size = VEC_LEN * sizeof(int);

// Fill A, B

// Create buffers in context.
cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, buf_size, nullptr, &error);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, buf_size, nullptr, &error);
cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, buf_size, nullptr, &error);

// Copy the data from host to device
error = clEnqueueWriteBuffer( cmd_queues[0], bufA, CL_TRUE, 0, buf_size, A.data(),
          nullptr, NULL, NULL);
error = clEnqueueWriteBuffer( cmd_queues[0], bufB, CL_TRUE, 0, buf_size, B.data(),
          nullptr, NULL, NULL);
```

Listing 2.6: create and fill opencl buffers

- **Create, compile, and extract *kernel*** OpenCL program usually is stored in a separate file and consists of kernels, helper functions and constant data. Kernel is

a function that executes on the device. It may use other functions in its body but the execution always start from kernel function. Before executing it on the devices, it should be compiled and transfered to the device. In listing 2.7, program source is filled with listing 2.8. First, the source is compiled and build on all devices of $platform[0]$. Afterwards, the kernel is extracted from compiled program.

```
char** source;
// Filling the source with kernel source code.

std::string kernel_func_name = "vecadd";

// Create program from the source
cl_program program = clCreateProgramWithSource(context, 1, source, NULL, &error);

// Compile & Build the program
error = clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);

// Create the Kernel
cl_kernel kernel = clCreateKernel(program, kernel_func_name, &error);
```

Listing 2.7: Compiling a program to build a kernel

```
__kernel__ void vecadd(int n_elements, const int* a, const int* b, int* c){

    int threadIdx = get_global_id(0);
    if(threadIdx >= n_elements) return;

    c[threadIdx] = a[threadIdx] + b[threadIdx];

}
```

Listing 2.8: Vector addition OpenCL kernel

- **Executing *kernel***: To execute a kernel on a specific device, a command should be submitted in the command-queue of the corresponding device. Furthermore, the *kernel programming model* should be specified. It defines how the concurrency model is mapped to the physical hardware. In OpenCL, each instance of the kernel executing function body is called *work-item*. The work-items group into *work-groups*. Work-items in a work-group can synchronize using barrier function, and both has access to shared memory address space. The number of work-items in a work-group is called *local work size*. The total number of work items needed to be executed is called *global work size* (Figure 2.7) . This execution model is mapped to device features. As an example, when this kernel is running on GPU like GTX 980, it is better to match the size of work-group to number of CUDA cores in SMM (128). Kernel running is not a blocking task. The host execution

path can be proceed while the kernel is running on device. If the results are required, the host should wait for kernel termination using one of the OpenCL provided mechanisms. In listing 2.9, the arguments of the kernel are set. As the kernel is going to be run on 1D array, the work size is 1D. The global work size is equal to number of elements in the array and the local work size is 128.

```
cl_int num_elements = VEC_LEN;

// Setting Kernel Arguments
error = clSetKernelArg(kernel, 0, sizeof(cl_int), &num_elements);
error = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufA);
error = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufB);
error = clSetKernelArg(kernel, 3, sizeof(cl_mem), &bufC);

// Defining the programming model
size_t local_ws = 128;
size_t global_ws = std::floor((VEC_LEN + local_ws − 1) / local_ws) ∗ local_ws;

// Executing the kernel on the device
error = clEnqueueNDRangKernel(cmd_queues[0], kernel, 1, nullptr,
            &global_ws, &local_ws, 0, nullptr, nullptr);
```

Listing 2.9: Vector addition OpenCL kernel

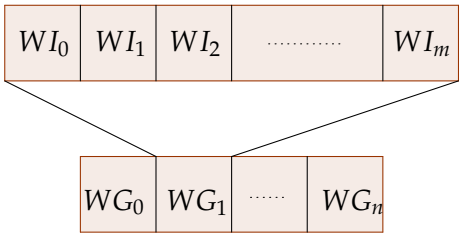


Figure 2.7.: **NDRange of work items** grouped into work-groups.

Like any other C/C++ application, the OpenCL allocated memories also should be clean up using its release functions. OpenCL also supports a general memory model which maps to the device's hardware memory during kernel execution. The device memory model is divided into four memory regions (Figure 2.8). *Global memory* which is visible to all work-item executing a kernel. Whenever the data is copied to device, it transfers to the global memory. Its size is equal to the dedicated GPU memory. *Constant memory* is also part of global memory. It contains the values does not change during kernel execution and used by all work-items simultaneously. *Private Memory* is the memory visible only to each work-item. This memory space practically maps

to registers, although arrays and spilled registers are mapped to off-chip memory. *Local memory* is shared among work-items of a work-group. It usually maps to on-chip memory. The latency of global memory is much higher than shared and private memories. Consequently, data access to shared and private memory is much faster than global memory.

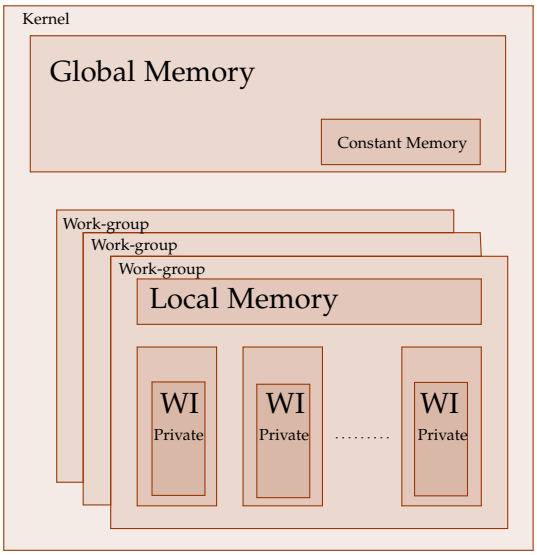
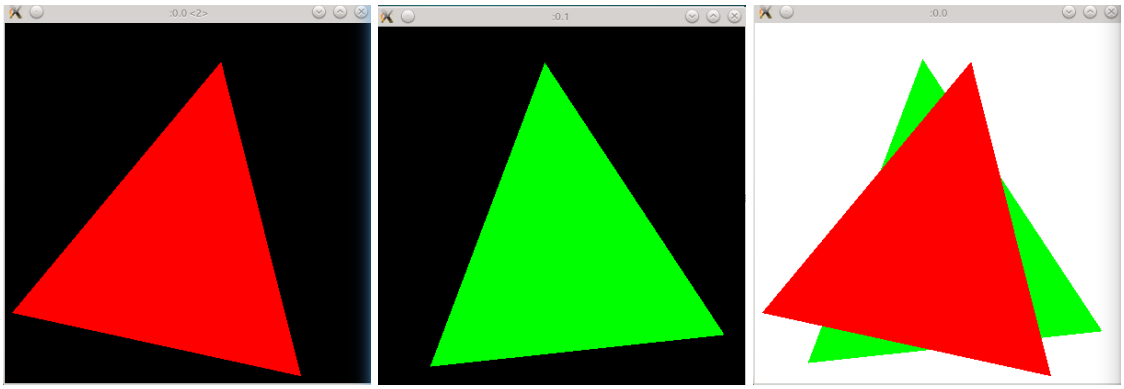


Figure 2.8.: **OpenCL memory model** consists of four different memory groups: Global-Memory which is usually mapped to off-chip memory. Constant memory which is part of global memory and contains constant data accessed by all threads. Private memory which contains the local variables in the kernel and usually mapped to registers. Local memory which is shared among a work-group's work-items; It usually mapped to on-chip memory.

In this master thesis, the goal is to not only compute the stream surface but also visualize it. The visualization is done with OpenGL library. This library has its own buffer types used for drawing. To render the streamsurfaces, the data should be read back to CPU, then copied into GL buffers. The read-back and copy adds a high overhead to streamsurface visualization. OpenCL provides a mechanism called OpenCL-OpenGL interoperability. It is possible to share a buffer among OpenGL and OpenCL library. Therefore, OpenCL can be used for computations and OpenGL can be used for rendering. The corresponding OpenGL context used for rendering should be declared during OpenCL context creation. The OpenCL-OpenGL interoperability should be supported on the computation device. Most of GPUs provide it. This feature does not exist on CPUs, therefore as discussed the data should be read-back, then

copied into GL buffer.



(a) GPU 0 (b) GPU 1 (c) Composition

Figure 2.9.: **Multi GPU rendering**. The left figure is rendered on GPU0, middle one rendered on GPU1, and the results are combined in right figure.

Two problem rises when multiple GPUs are used in a computer. First, when using OpenCL-OpenGL interoperability, the context cannot be shared among more than one GPUs, even if they are produced by the same vendor. Therefore, one OpenCL context should be created for each GPU. Second, each OpenGL context can use only one GPU for rendering at the time. Consequently, each GPU requires a OpenGL context. In windows, the responsible GPU for an OpenGL context is determined by the operating system. Therefore, creating OpenGL context on a device is not possible except for Nvidia Quadro GPUs. Using Linux, it is very tricky but possible to handle it.

When using more than one gpu, compositing the rendered results (framebuffers) is also challenging because the resources are in different contexts. As long as, contexts exist on two different GPUs, sharing buffers and textures among contexts is not possible. The simplest but with highest workload is to transfer the buffer or texture from other GPU to GPU rendering final image. The best option is to use extensions of OpenGL which provide fast copying textures between two contexts. Figure 2.9 shows a green triangle is render on a Nvidia Geforce GTX 980 and a red one is rendered on a Nvidia Gefore GTX 970, the results are combined in another window.

In runtime, OpenCL is running on CPU by creating a thread on each CPU core. These threads are passed to thread manager unit of each core to be enqueued for execution. To increase performance, the compiler tries to vectorize the code to increase the performance by using *Single Instruction Multiple Data* (SIMD) instructions like SSE, AVX, and etc. These instructions are used to run one instruction on multiple data simultaneously. Also, the developer can vectorize the code explicitly. GPU architecture

is significantly different with CPU. The GPU benefit from wider SIMD units, supports heavy multi-threading, and programmable physical memory buffers e.g. L2 cache. It is more convenient to refer GPU threads as *wavefront* (AMD terminology) or *warp* (Nvidia terminology) which consists of multiple threads on the hardware. As an example, a warp on Nvidia Geforce GTX 980 is made of 32 threads and a wavefront on AMD Radeon R9 290X consists of 64 threads. Wavefront/warp is the most basic unit executing on the GPU which is assigned to a SIMD unit to be executed.

As described, both CPU and GPU execution models are based on SIMD units. SIMD units are performing efficiently if all the threads in wavefront/warp execute the same instructions, simultaneously. But there are circumstances they do not execute the same instruction e.g. when a warp is executing *if-else* instruction and not all of them fulfill the branch condition. In this case, the threads running the else part must wait for threads running the if section, and vice versa. It is called *divergance* in code and cause *incoherency* in executing the instructions. Consequently, the performance will drop significantly. Furthermore, if threads in a wavefront/warp issue memory instruction which result in different memory accesses, the instruction execution latency will increase and the performance will drop. It is called incoherency in the data. Both incoherency in instruction and data result in performance drop and should be avoided as much as possible.

2.3. Related Works

The most related work which this master thesis is built upon is [**6378974**] which has proposed an algorithm for generating streamsurface computation in a distributed memory system. If two adjacent streamlines diverge far from each other, a new seeding point is inserted on the seeding curve. It was initially introduced by [**235211**]. Additionally, the parallel particle algorithms hat are used for streamsurface computation are investigated. Many papers are published on parallel streamline/streamsurface computation on GPU. However most of them are sampling the simulation results on a regular grid (3d texture). [**5290737**] presents a real-time technique for computing and rendering streaksurfaces on the GPU. [**Schirski:50085**] uses the neighboring of the simulation mesh's grid information for particle tracing into a grid of tetrahedral cells. The initial cell locating is performed on the CPU using a kd-tree. The particle is integrated using the tetrahedral cells neighboring information. If the next point was not in any of the neighboring cells (due to large step size), the acceleration structure on CPU is used to locate the cell containing the particle position.

In this master thesis, the streamlines are integrated in an unstructured grid. In the recent years, interesting papers are presented on using acceleration structures for

speeding up cell look-up. [**Wilhelms:1992:OFI:130881.130882**] uses octree for looking up cells for isosurface extraction. In this approach each octree's leaf nodes stores the cell indices overlapping it. The octree is subdivided until the number of cells in the leaf is less than a predefined maximum cells per leaf. The cell look-up is done by starting from the root node, and traverse the children until a leaf is visited or the point is not inside children volume. An implementation of this method can found in VTK library. [**5613496**] presents a flexible and memory-efficient accleration structure called cell-tree. The cell-tree is a cell partitioning approach which is used for doing interpolation in the simulation mesh. In this approach, cells are partitioned based on a defined cost function along one axis. Partitioning is done until the number of cells in a node is less than a predefined maximum cells per leaf. Cell-tree traversing starts from the root node. The nodes are traversed if the point is in the node's interval along the split axis. If the point is in both children intervals, one is kept in stack and the other one is traversed.

3. Acceleration Structures

3.1. Motivation

(a) Spatial Partitioning (b) Object Partitioning

Figure 3.1.: **Subdivision Approaches**. In the left figure, the primitives are partitioned into two disjoint sets but the areas are overlapping. In the right figure, the space is partitioned into two non-overlapping areas but one of the primitives overlaps both partitions.

Nowadays, *acceleration structures* are the heart of ray tracers. Without them each ray has to be tested against all primitives in the scene, even if the ray is traveling far from them. The main goal of using acceleration structures is to reduce the number of ray-primitive tests to find the closest intersection point. Two requirements are necessary for acceleration structures are: First, an acceleration structure should be able to skip groups of primitives. Second, it should order the search process so the nearest intersected primitive is found first. There are two approaches to acceleration structure: *spatial partitioning*, and *object partitioning*. Spatial partitioning means dividing the 3d space into regions which store references to primitives overlapping them. If the ray intersects with the region, a ray-primitive test is done for all contained primitives. Kd-trees[KDTREEREF] and grids which were briefly described in the previous chapter are examples of spatial partitioning structures. Object partitioning is progressively dividing the primitives into smaller set of primitives. Bounding volume hierarchy (BVH)[BVHREF] and bounding interval hierarchy (BIH)[BIHREF] are two examples of object partitioning structures. There is a *cost function* which determines object splitting aspect. Figure 3.1 shows a scene divided spatially and by object. Both of these

approaches are successful in different circumstances and there is no specific reason to prefer one over another.

There are many similarities between streamline tracing and ray tracing. For example, tracing streamlines seems like tracing curved rays. As it is discussed in the previous chapters, streamlines are computed by directing a particle starting from the seeding point through a flow field. To look up a velocity vector, locating the simulation cell containing the sample point is necessary. To increase the streamline computation performance, acceleration structures can be utilized to skip multiple groups of cells in order to minimize the number of unnecessary point-in-cell tests. While streamline computation and ray tracing are similar, they have substantial differences. Streamline computation requires looking up the velocity vector at a specific point of the flow field while raytracing tests for ray-primitive intersection. These differences can cause different cost functions in acceleration structure computations.

This chapter is mainly about accelerating the cell lookup. It will present how to construct and traverse grid, BVH, Kd-tree, and cell-tree acceleration structures for the representation of an unstructured (simulation) grid. It is assumed that the unstructured simulation grid is defined as a list of N cells. For each cell c_i, $(i = 0, ..., N-1)$ its axis-aligned bounding box B_i and its center (*centroid point*) C_i are defined. This definition for unstructured simulation grid is very general and it does not provide any information about the topology of the cells. Additionally, there is a section about using the cell neighboring information to compute the streamlines without generating any acceleration structure.

3.2. Grid

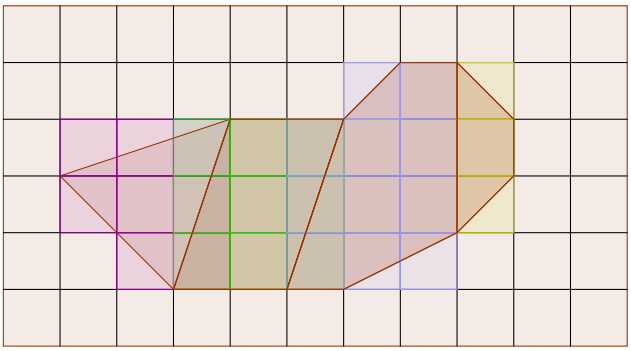


Figure 3.2.: **Grid acceleration structure**. It partitions the space into equal sized boxes called voxels. Voxels keep a record of cells overlapping with them.

A regular grid is a structure which divides the space into equal sized boxes called *voxels*[**Pharr:2010:PBR:1854996**]. Each voxel stores the references to cells whose bounding boxes overlap with the voxel. Figure **??** shows a simple representation of a grid.

To look up the cell containing the sample point, the grid's voxel surrounding the sample point is determined; all cells associated to the voxel are tested against the sample point to find the containing cell. Unnecessary intersection test are reduced significantly by considering only cells reside in the voxel containing the sample point.

A grid's construction is very fast. Furthermore, testing sample point against the voxel is simple. This simplicity is a double-edged sword. When cells are not of the same extent, and are distributed non-uniformly in space, the performance of the grid may drop. As an example, assume a flow field's unstructured simulation grid with a non-uniform cell density distribution. In this case if the acceleration grid is constructed coarsely, many cells falls into a voxel, whereas a fine grid is used, many voxels store references to large cells. This problem is known as the "*teapot in a stadium*" problem in raytracing because it is like having a high resolution Utah teapot mesh in a low resolution stadium mesh.

The main problem of this acceleration structure is lack of any cell distribution adaptation mechanisms. Unstructured simulation grids with cells of different types and sizes are not unusual in scientific visualization, which can reduce grid acceleration structure performance when tracing streamlines.

Grid acceleration structure construction is a general method which take the cells' bounding boxes (AABBs) as input. To distribute the cells in the grid uniformly, the number of voxels is chosen proportional to the number of cells. Although increasing the number of voxels decreases the number of cells per voxel which results in efficiency improvement, it will also increase the grid size in the memory. In contrast, a low voxel count will result in a low performance acceleration structure.

If N is equal to the number of cells, and the extents of surrounding bounding box are stored as $extent[i]$. $\sqrt[3]{N}$ is an appropriate approximation for number of voxels along the maximum extent of the flow field's AABB. The number of voxels along other dimensions is determined based on the ratio of other dimensions to the maximum one. In practice, $a\sqrt[3]{N}$ is used as the number of voxels along maximum extent. The value of a multiplier is determined by experiments. In this master's thesis $a = 3$ is used, as suggested in [**Pharr:2010:PBR:1854996**].

Associating cells to voxels is done by adding their references to all voxels that overlaps the cell's bounding box. The Listing 3.1 represents the algorithm to find the overlapping voxels. It is done by calculating the voxels containing the minimum and maximum points of cell's AABB. All the voxels between these two voxel indices are storing a reference to the corresponding cell. This process has to be iterated for all cells of the flow field.

```
// c: Current cell
for(int axis = 0; axis < 3; axis++){
  vmin[axis] = position2Voxel(c->bbox.min[axis]);
  vmax[axis] = position2Voxel(c->bbox.max[axis]);
}

for(int z = vmin[2]; z <= vmax[2]; z++)
  for(int y = vmin[1]; y <= vmax[1]; y++)
    for(int x = vmin[0]; x <= vmax[0]; x++)
      voxel[x][y][z].associate_cell(&c);
```

Listing 3.1: **Associating cells with the voxels of the grid**. The overlapping voxels are computed by looking up the voxels containing the mimimum and maximum point of the cell's bounding box. All the voxels between the minimum and maximum voxel index also store a reference to the cell overlapping with them..

A grid is traversed by locating the voxel that the sample point resides in. The point should be tested against all cells contained in the voxel. The point-inside-cell test is done only if the point lies inside the cell's bounding box. The point-inside-cell test is done by calculating the point's parametric coordinates (See Section 2.1). If all of them are between zero and one and sum of them is equal one, the point is inside the cell. As cells are not overlapping each other, once a cell containing the sample point is found, the look up operation can be stopped. The parametric coordinates can be used to interpolate velocity or any other attribute in the cell.

While the grid acceleration structures used in raytracing and cell locating are constructed similarly, there is a big difference between finding the closest intersection point in ray tracing and cell lookup for streamline computation. The first one results in a list of voxels through which the ray passes. The primitives contained needs to be tested against the ray from front to back until the primitive intersecting with the ray is found. Here, voxels may store the same primitive multiple times which results in multiple intersection of the same primitive. The second one's outcome is a voxel whose its associated cells should be tested against the sample point. Consequently, cell locating expected to offer better performance compared to finding the closest intersection point with a ray.

3.3. Bounding Volume Hierarchy

A bounding volume hierarchy (BVH)[BVHREF] is a hierarchy of disjoint subsets of cells which is structured based on a cell-set partitioning approach (in contrast to a spatial partitioning approach). The hierarchy is basically a binary tree which consists

of two node types: The interior nodes store a bounding box around the contained cells. The leaf nodes keep track of cells. The idea behind using a hierarchy is to skip a subtree beneath a node when the sample point is not inside its bounding box. As BVH construction is based on a cell-set partitioning approach, each cell appears only once in the hierarchy. The amount of memory required for storing a BVH is bounded by the number of cells inside the flow field. If the flow field contains N cells, the memory required to store the acceleration structure is bounded by $2N - 1$ BVH nodes storing one cell per leaf node (N leaf nodes), and interior nodes for building the hierarchy ($N - 1$ interior nodes).

The BVH can be constructed using two different methods: top-down, and bottom-up. The first method is started by initializing a BVH node with the whole cells of the flow field. Cells are partitioned into two disjoint subsets and two new BVH nodes are created containing them. The partitioning step continues until an appropriate hierarchy is achieved. The bottom-up method starts by grouping cells into leaf nodes. The hierarchy is constructed by continuously merging the leaf, and interior nodes together [PHYSICREF]. Although the hierarchy from the bottom-up method is the best possible BVH, its construction is too costly. Therefore, the top-down approach is chosen for implementation in this master's thesis.

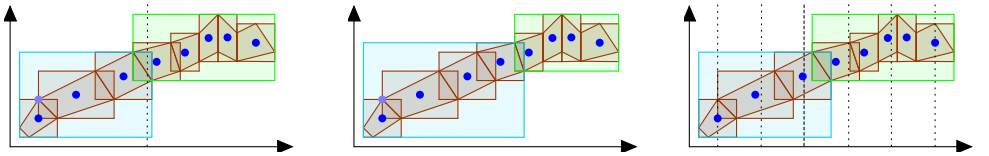
BVH acceleration structure construction for the unstructured simulation grid consists of two steps: First, the BVH is built using a recursive procedure. Then, its representation is converted to a pointerless, linear, and compact array of nodes. Although the BVH can be built directly into a linear, and compact representation, debugging needs more effort. Therefore, the two step approach is implemented here. A BVH is a hierarchy of nodes where each of them represents a set of cells, and a bounding box surrounding them. Building it starts by computing the axis-aligned bounding box around all cells of the flow field. The bounding box accompanying the set of cells make the first node of the BVH. The associated cells with each node are partitioned into two disjoint sets and new BVH nodes are created with them. This procedure continues recursively until the number of cells is less than or equal to *maximum number of cells per leaf* (*max_cells_per_leaf*), or cost of partitioning the node is greater than testing the sample point with the all cells in the BVH node. The *max_cells_per_leaf* is used to control the depth of the tree, and the memory usage of the BVH. Therefore, no maximum BVH depth or memory usage limit definition is required. Listing 3.2 shows the code for building a BVH node. It decides if a leaf node can be created from cells inside a region, or if they should be partitioned into two other nodes.

```
BVHNode* construct_bvh(std::vector<Cell> cells, uint start, uint end, std::vector<Cell>& ordered_cells) {
  BVHNode* node = new BVHNode;
  BBox bbox;
```

```
  // Calculate the bounding box around cells in current region
  for(uint c = start; c < end; c++) bbox.extend(cells[c].bbox);

  uint num_cells = end − start;
  if(num_cells <= max_cells_per_leaf){
    // The cells belong to the leaf node are added to the ordered_cells list
    node−>initLeaf(...);
    return node;
  }
  else{
    uint dim = cell_split_dim();
    uint splitIdx = cell_split_idx(start, end);

    // if splitting cost is greater than testing
    if(cost(splitIdx) >= (end − start) ∗ t_incell){
          node−>initLeaf(...);
          return node;
    }

    node−>initInterior(
      dim,
      construct_bvh(cells, start, splitIdx+1, ordered_cells),
      construct_bvh(cells, splitIdx+1, end, ordered_cells)
    );
    return node;
  }
}
```

Listing 3.2: **Bounding volume hierarchy (BVH) construction** pseudocode. It generates a sub-tree from cells in range $[start, end)$ of cells array. It checks if the number of cells inside the sub-tree is less than or equal to $max\_cells\_per\_leaf$, then creates a leaf node. Otherwise, it will create an interior node and split cells into two partitions and make two new child nodes.

In general, there are $2^N - 2$ possible ways to split a set of cells of a unstructured simulation grid into two non-empty disjoint subsets. To make the acceleration structure as efficient as possible, finding an appropriate cell partitioning method is important. In practice, the choices for split planes usually are limited to the edges of cell's bounding boxes along each coordinate axis ($6N$ candidates as each bounding box has two edges along each coordinate axis). Furthermore, the BVH node splitting is done along the coordinate axis called *split axis* along which the extent of centroid points is largest along it. Since too much overlap of sibling BVH nodes results in frequently traversing both children during BVH traversal, the *split point* should be selected at a position where the overlap of bounding boxes of two cell subsets is minimized. Figure 3.3 shows three different splitting methods: *midpoint*, *equally-sized subsets*, and *cost function*. In



(a) Mid point along split axis (b) Equal cell count (c) Cost functions for different buckets

Figure 3.3.: **Three different cell splitting methods**. In the left figure, the center of the split axis is chosen as the splitting point. In the middle one, cells are partitioned into equal sized subsets of cells. In the right one, the bounding boxes around cells are split into equal sized buckets. A cost function is calculated for edges of the buckets and the minimum edge with minimum cost function is chosen as the split position. The centroid of cells are tested against the split axis during cell partitioning.

all these methods, a split point along a split axis is chosen; If the centroid point of a cell's bounding box is below or on the split point along the chosen split axis, the cell belongs to left child otherwise its reference is stored in the right child. One of the simplest partitioning approaches is the midpoint. In this approach, the center of axis-aligned bounding box which contains the centroid points of associated cells to the BVH node is chosen as split position. It is used to partition cells into two disjoint subsets of cells. Figure 3.3a shows a grid of cells partitioned by the midpoint method. Another splitting approach is partitioning cells into two equally-sized subsets along split axis,i.e., classifying the first half of cells along split axis in left child and the other half in right child. Figure 3.3b shows an example of partitioning cells by splitting them into equal-sized subsets.

For normal regular grids, the midpoint and equally-sized partitioning approaches can work well, but for unstructured simulation grids, the result BVH usually leads to poor performance of acceleration structure. Raytracing algorithms benefit from partitioning method which usually end up with a optimized BVH called *Surface-Area-Heuristic* (SAH) [PBRTREF] which calculates the split position by minimizing the cost of partitioning the BVH node's associated cells. The estimated cost is defined as a *cost function* which is defined as follows:

$$c(A,B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i)$$

where $c(A,B)$ denotes the cost function for partitioning the cell set into sets A and B, t_{trav} is the time to traverse the interior node, $t_{isect}(x)$ is the time to intersect ray with primitive x, p_A and p_B is the probability of a ray passing through partition A or B, and

N_A and N_B are the number of primitives in partitions A and B. Estimating p_A and p_B is usually done using ideas from geometric probability theory. If two convex volumes V_0 and V_1 are assumed where convex volume V_0 contains convex volume V_1, it can be shown that the conditional probability that a random ray passing through convex volume V_0, will also pass through volume V_1 can be calculated by dividing the surface area of volume V_1 by the surface area of volume V_0. Consequently, p_A and p_B can be calculated by dividing the surface area of the axis-aligned bounding box around A or B by the surface area of the parent node.

A similar cost function can be proposed for constructing an efficient BVH for cells of a flow field. As a point-in-cell test is performed in this method instead of a ray-primitive intersection, t_{trav} is replaced with t_{incell} which is the time for doing a point-inside-cell test. It is assumed that t_{incell} is the same for all cell types. The main differences between cost functions used for raytracing and streamline tracing are originating from their application. As the BVH constructed for cells of a flow field is eventually used for cell lookup, p_A and p_B are calculated differently. For two bounding boxes A and B where A contains B, it can be shown that the conditional probability of a random sampling point residing inside bounding box A and residing in bounding box B can be calculated by dividing the volume of B by the volume of A. Therefore, the cost function for BVH construction is defined as follows:

$$c(A,B) = t_{trav} + \frac{vol(A)}{vol(currnode)} N_A t_{incell} + \frac{vol(B)}{vol(currnode)} N_B t_{incell}$$

As already mentioned, the split axis is chosen along the coordinate axis with the largest centroid points extent. Although the split position can be limited to the edges of cell bounding boxes ($2N$ choices for edges of bounding boxes along split axis), still it is very costly to calculate the cost function for all the edges specially for very large datasets. To make the BVH construction faster, the bounding box of the node is divided to equal extent, and only the boundaries of buckets are considered as split position candidates. Figure 3.3c shows the boundaries of buckets. The bucket edge with minimum cost function value is chosen for partitioning cells. While this approach is not constructing the most effective BVH, the performance of the result BVH s comparable to the BVH tree constructed using a bottom-up approach [**PHYSICREF**][**PBRTREF**].

The result BVH from the previous step is a tree with nodes containing pointers to their children. To improve cache and memory performance, this representation is converted into two compact linear arrays of nodes, one stores the interior nodes, and the other is an array of leaf nodes. The separation into interior and leaf node arrays aims to reduce memory size of the BVH. Listing 3.3 shows the definition of a BVH node in the construction step, and BVH interior and leaf nodes in the linear and compact representation. The Linear BVH is a depth-first representation of the tree where the left

subtree is stored first and the right subtree is following it. While it is implicitly known that the left child is stored immediately after a node due to the depth-order structure, it is still required to store the left child's index explicitly to keep track of it if it is a leaf node; because they are stored in different arrays. The leaf node indices are stored as a negative number in the interior node's child field to distinguish it from the interior node array's index. Therefore, if a child node index is negative, it is referencing to an index in leaf array which its index can be calculated with $1 - index$.

```
struct BVHBuildNode {
  AABB bbox;
  BVHBuildNode* child[2];
  uint32_t splitAxis, firstPrimOffset, nPrimitives;
};

struct LinearBVHInteriorNode {
  // bounding Box around the left and right child
  AABB leftBound, rightBound;

  // child index in array
  // If negative, the child is a leaf node,e.g., -1 -> 0 (-index - 1), -2 -> 1 (-index - 1), ...
  int32_t children [2];

  // splitAxis: x = 0, y = 1, and z = 2
  uint8_t splitAxis;
};

struct LinearBVHLeafNode {
  // offsets into the point array
  // this leaf includes voxels [low, high)
  int lowIdx, highIdx;
};
```

Listing 3.3: **BVH struct definitions which are used during construction and traversal**. BVHBuildNode is used during the construction where the BVH is stored in pointer representation. In both representations, the interior node is storing the split axis, and references to the children, while the leaf nodes keep record of contained cells. In the linear compact representation, the interior nodes store the bounding boxes of both children to reduce memory reads and increase cache performance.

BVH traversal starts from the hierarchy's root node. It basically goes through the hierarchy by testing the point against bounding boxes of the nodes and cells. Listing 3.4 shows the BVH traversal pseudocode. BVH traversing starts by testing the sample point against the root node and continues by going down in the hierarchy. For an arbitrary node with index *nodeNum*, if *nodeNum* is negative it is a leaf node and the

point should be tested against all associated cells. If *nodeNum* is positive it is an interior node; therefore the sampling point should be tested against bounding boxes of children nodes. If the sampling point is in one of the children's bounding boxes, the traversal path is clear. If the point is in both of them, one should be chosen for traversing in next step, and the other one should be kept on a stack for testing afterwards in case the cell containing the sample point is not found. Experiments shows if the children where the sampling point is closer to its bounding box center is chosen for traversal first, the number of tests to find the relevant cell will decrease. In the traversing step, whenever the path for going down the BVH stops e.g., when the sample point is not in any of the children's bounding boxes, or when the sample point is not inside any of the cells of the leaf node, the next node to restart the BVH traversal is poped from the stack. This procedure continues until the appropriate cell is found, or all the possible cells are tested against the sample point and no cell containing the sample point is located.

```
// bvhtree: the BVH constructed for an unstructured simulation grid
// s: sampling point
// returns the cell contatining the sampling point
Cell* findSurroundingCell(BVHtree* bvhtree, Point* s) {
  int nodeNum = 0;
  Stack stack;
  // test if the sample point is inside the bounds of BVH.
  if(!inside(bvhtree->bounds, s)) return nullptr;
  while(true){
    // if the current node is an interior node
    if(nodeNum >= 0){
      const LinearBVHInteriorNode* node = &bvhtree->m_interior_nodes[nodeNum];
      const bool insideLeftChild = node->leftBound.contains(*s);
      const bool insideRightChilld = node->rightBound.contains(*s);
      if(insideLeftChild && insideRightChilld){
        const float d0 = (node->leftBound.center() - *s).length();
        const float d1 = (node->rightBound.center() - *s).length();
        if(d0 <= d1){
          stack.push(node->children[1]);
          nodeNum = node->children[0];
        } else {
          stack.push(node->children[0]);
          nodeNum = node->children[1];
        }
      } else if(insideLeftChild && !insideRightChilld){
        nodeNum = node->children[0];
      } else if(insideRightChilld && !insideLeftChild){
        nodeNum = node->children[1];
      } else {
        if(stack.empty() == 0) break;
        nodeNum = stack.pop();
```

```
      }
    // current node is a leaf node
    } else {
      const LinearBVHLeafNode* node = &(bvhtree->m_leaf_nodes[-nodeNum - 1]);
      for(int i = node->lowIdx; i < node->highIdx; i++)
        if( insideCell( i, glm::vec3( s ) ) ) return &cells[cell_index];
      if(stack.empty()) break;
      nodeNum = stack.pop();
    }
  }
  return nullptr;
}
```

Listing 3.4: **BVH traversal algorithm**. Locating a cell containing the sample point is done by first testing the bounding box around the BVH. The traversal continues by testing the sample point against the children's bounding boxes for interior nodes, and testing it against cells associated with leaf nodes. A stack is used to keep track of not tested nodes when the sample point is inside both children. This procedure continues until the cell containing the sample point is found or all the possible traversal paths are tried and no appropriate cell is found.

3.4. Kd-tree

As discussed in Section 4.2, grid acceleration structures have a low performance when they are constructed for a flow field defined with an unstructured simulation grid. *Binary space partitioning* (BSP) structures are irregularly adaptively partitioning the space into two regions. BSP construction starts with an axis-aligned bounding box around the entire flow field. It is recursively partitioned until the number of cells overlapping with it are small enough, or the structure's depth is greater than a predefined maximum depth value. Spatial partitioning is done using a splitting plane which can be placed arbitrary in the region. The hierarchy can have different depth levels in various parts of it,i.e., Parts of the 3d space contains less number of cells are partitioned less while parts of it containing more number of cells are partitioned more.

Two variants of BSP trees are: *Kd-tree*, and *octree*. Kd-trees split the region using a plane perpendicular to one of the coordinate axes. Kd-tree construction is a top-down recursive algorithm, which means a tree is constructed by iteratively partitioning the bounding box enclosing the flow field's cells into two sub-regions. Each sub-region keeps references to cells overlapping it. The partitioning terminates if the number of cells overlapping a region is less than a specific threshold or the depth of the tree is greater than a predefined maximum depth. As [PBRTREF] suggests, $8 + 1.3 * \log_2 N$ is

a good approximation for as maximum tree depth, where N is number of cells in the flow field.

One of the challenging problems of Kd-tree construction is choosing the right split plane. It must be carefully chosen to make the tree traversal as efficient as possible. The cost function used for BVH construction can also be applied to the Kd-tree construction problem. Kd-tree's correct ratio between t_{incell} and t_{trav} has a huge impact on Kd-tree's performance and size in the memory and should be adjusted properly.

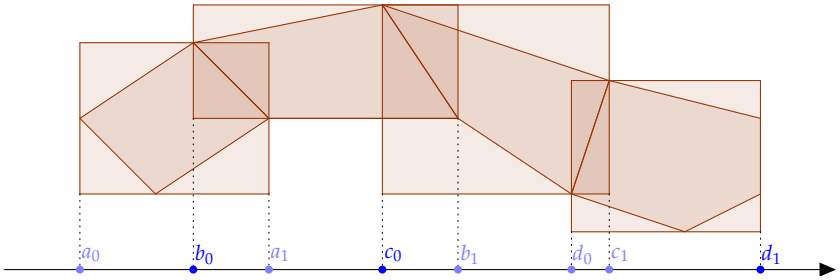


Figure 3.4.: **Kd-tree split plane** is chosen by computing the cost function for all bounding boxes edges. The edge with minimum cost function value is chosen as position of the split plane. If the minimum cost function value is greater than testing the sample point against the node's associated cells, instead of splitting it, a leaf node containing all cells is created.

Similar to the BVH, a Kd-tree consists of leaf and interior nodes. Interior nodes are split by a plane perpendicular to one of coordinate axes. All of them have two children. The split plane is chosen where the cost function has minimum value. It can be proven that the minimum cost is attained on one of the cells' AABB faces. Therefore, the number of choices for a split plane is reduced to $6N$ faces of cell bounding boxes. Thus, there are $2N - 2$ possible split planes perpendicular to each coordinate axis. There is an effort to split the region along the maximum extent to generate cubic shape sub-regions (to not split repeatedly along one axis). Therefore, the candidates perpendicular to the coordinate axes along the maximum extend is evaluated first. If the appropriate split plane was not found, the other axis are tried. Listing 3.5 presents how the cost function is used to find the best possible split plane among all the planes perpendicular to the coordinate axis (*axis*). Cost functions can be calculated in a for-loop very quickly by sweeping across the projection of the cells' bounding boxes along evaluation coordinate axis. Each bounding box has two edges on the coordinate axis: the *start* edge which is met first during sweeping, and the *end* edge which is met later during sweeping. Figure 3.4 shows how the cell's bounding boxes are projected on the evaluation axis. As it is shown each of the bounding boxes has two edges on the coordinate axis which

the start edge is shown with a lower index and the end edge has greater index. The *edges* array keeps track of all bounding boxes edges, their type (start or end), and their split position in 3d space.

Looking up for the best split axis starts from the left and the cost function for each edge is calculated during sweeping from left to right. In the beginning, all cells are above the split position ($num_{above} = num_{primitives}$). While iterating through edges, if an edge is a starting edge, the number of cells below the split plane is increased, and if an ending edge is observed the number of above edges is increased. By determining the number of cells in each side of a candidate split plane, and the probability of sampling the point in each sub-region, the cost function can be calculated. Eventually, by calculating the cost function for all edges along a specific coordinate axis, the best possible split plane with its partitioning cost is attained. If the minimum cost is greater than not splitting the region cost (*not_split_cost*), a leaf node containing all cells is generated instead of splitting the region into two sub-regions.

```
// best_cost: best cost value
// best_axis: best cost attained with splitting plane prependicular to this coordinate axis.
// axis : test split planes prependicular to coordinate axis.

void find_split_plane_along_axis(Flowfield* flowfield, int axis) {
  float not_split_cost = incell_cost * num_cells;
  Vec3f dims[3] = flowfield.bbox.max_point - flowfield.bbox.min_point;
  float total_volume = dim[0] * dim[1] * dim[2];
  int num_below = 0, num_above = num_primitives;
  for(int i = 0; i < 2 * num_primitives; i++){
    if(edges[axis][i].type == END_EDGE) --num_above;
    float edget = edges[axis][i].t;
    float below_volume = d[OtherAxis0] * d[OtherAxis1] * (edget - bounds.min);
    float Above_volume = d[OtherAxis0] * d[OtherAxis1] * (bounds.max - edget);
    float p_below = below_volume / totalVol;
    float p_above = above_volume / totalVol;
    float eb = (num_above == 0 || num_below == 0) ? empty_bounus : 0.0f;
    float cost = trav_cost + incell_cost * (p_below * num_below + p_above * num_above);
    // Update the best_cost, best_axis, and store the splitting plane position.
    if(edges[axis][i].type == START_EDGE) ++num_below;
  }
}
```

Listing 3.5: **Finding the best possible split plane** perpendicular to a coordinate axis using the cost function. In this method, the cost function for all of the cells' bounding boxes edges are calculated to find the best possible split plane.

The data representation of a Kd-tree in the memory is similar to the BVH. It is stored in depth-first order, therefore the left sub-tree of a node is stored completely immediately after its parent node, and right after that the right sub-tree is stored.

Listing 3.6 shows how the nodes of the Kd-tree are declared. There is an effort to keep the size of the node 8-bytes to align four Kd-tree nodes in a CPU cache line. This representation is used both for interior and leaf nodes. The first 4 bytes of the Kd-tree node is either storing the split position for interior nodes, or the cells inside a leaf node. When the leaf node contains just one cell, instead of a pointer to array of cell indices, the cell index is stored directly. To differentiate between interior nodes split coordinate axis and leaf nodes, the first two bits of the second 4 bytes of a Kd-tree node are used that contain 00, if the split plane is perpendicular to X coordinate axis, 01 if it is perpendicular to Y coordinate axis, 10 if it is perpendicular to Z coordinate axis, and 11 if the node is a leaf node. The other 30 bits are used for storing the number of cells overlapping with a leaf node, or the right child of the interior node (the left child node is defined to follow immediately after the parent node implicitly).

```
struct kdtreeNode{
  union {
    float split; // Interior
    uint32_t one_cell; // Leaf
    uint32_t* cells; // Leaf
  };
  union {
    uint32_t flags; // Both
    uint32_t n_cells; // Leaf
    uint32_t right_child; // Interior;
  };
};
```

Listing 3.6: **Kdtree node definition**. The first two bits of the flag store the information about the node type, and the split axis for interior nodes. Additionally, the interior nodes store the split position and the right child (the left child is stored right after the parent as the Kd-tree is stored linearly in depth-first order). The leaf node keep record of overlapping cells and their number.

Traversing a Kd-tree to look up the cell containing the sample point is simpler than testing intersection of a ray and a Kd-tree. Listing 3.7 shows the code for locating the cell containing the sample point. As the Kd-tree is a space partitioning structure, the bounding boxes of nodes are not overlapping; therefore in contrast to the BVH traversal algorithm, a stack is not required for Kd-tree traversal. Traversal is done by starting from the root of the tree and testing the sample point against the bounding box of the root node. If the point is in the root's bounding box, the sample point is tested against the bounding boxes of the left and right children. It is either inside the left or right child bounding box. The Kd-tree is traversed using this procedure until finding a leaf node. Cells of the leaf node are tested against the sample point to locate the cell containing the sample point.

```
// kdtree: the kdtree constructed for a dataset
// s: sampling point
// returns the cell contatining the
Cell* findSurroundingCell(Kdtree* kdtree, Point* s) {
  KdtreeNode *node = kdtree->root;
  if(!node->bbox.contains(s)) return nullptr;
  while(true){
    if(node->type == INTERIOR_NODE){
      if(node->left_node.contains(s)) node = node->left_node;
      else if(node->right_node.contains(s)) node = node->right_node;
      else return nullptr;
    } else if(node->type == LEAF_NODE){
      for(size_t i = 0; i < node->cells.size(); i++)
        if(node->cells[i].contains(s))
          return node->cells[i];
      return nullptr;
    }
  }
}
```

Listing 3.7: **Cell locating using Kd-tree acceleration structure**. This procedure is used to locate the cell containing the sample point utilizing the Kd-tree acceleration structure. The inputs to the function are a pointer to kdtree acceleration structure, and sample point s. It returns a pointer to cell contatining the sample point. It return a null pointer when no cells are found.

3.5. Cell-Tree

The cell-tree [CELLTREEREF] is an acceleration structure built based on bounding interval hierarchy (BIH). It is structured using the cell-set partitioning approach. The bounding box around the whole cells of the flow field are recursively partitioned into two disjoint sets of cells using two split planes perpendicular to one of coordinate axis. Essentially, the construction and traversal algorithms for cell-tree are very similar to BVH; It is the partitioning method of cell-tree which makes it distinguished from BVH. The hierarchy of the cell-tree consists of two node types: interior, and leaf nodes. The first node types store indices of child nodes, split axis, and the left child's associated cells maximum extent and right child's minimum extent in split axis direction. Negative child index (C_i) refers to nodes in the leaf nodes array. The index in the leaf nodes array can be calculated by $-C_i - 1$. The latter node type keeps record of cells which their centroids are inside the leaf node's boundaries. Listing 3.8 shows the definition of cell tree's interior and leaf structures.

```
struct LinearCellTreeInteriorNode {
```

```
  // Left child maximum boundary, and right child minimum boundary in split axis direction.
  float Lmax, Rmin;

  // child index in array
  // If negative, the child is a leaf node,e.g., − 1 −> 0 (−index − 1), − 2 −> 1 (−index − 1), ...
  int children [2];

  // splitAxis: x = 0, y = 1, and z = 2
  uint8_t splitAxis;
};

struct LinearCellTreeLeafNode {
// offsets into the point array
// This leaf includes voxels [low, high)
int lowIdx, highIdx;
};
```

Listing 3.8: **The cell-tree's interior and leaf node structure definitions**. The interior nodes stores the minimum and maximum boundary extents of the left and right chid node, the indices to left and right nodes, and the axis which the nodes are split about. The leaf nodes store the minimum and maximum indices of the containing cells.

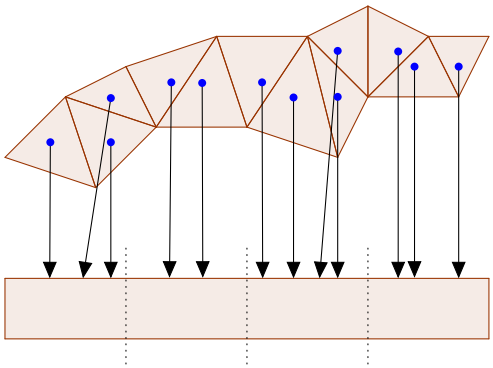


Figure 3.5.: **Cell-Tree**. **The cell-tree partitioning algorithm**. It divides the node's boundary into several buckets. The cost function is calculated for edges of buckets. The edge with the minimum cost function value is used for partitioning the cells into two subsets. The cells which their centroids are in the below or on the splitting plane is associated to the left child, and the ones which are above the splitting plane is associated to the right child.

The construction algorithm starts by storing all of the flow field's cell references in

a cell-tree node. Every cell-tree's node is either partitioned into two children, or is stored as a leaf node. A leaf node is initialized if the number of references to cells associated with a cell-tree's node is less than or equal to the maximum number of cells per leaf node, or the cost of splitting the node is more than testing the sample point against all cells associated with the node. The cells associated with a leaf node are reordered sequentially in the unstructured simulation grid structure to make the leaf node structure simple; Therefore, it can only keep the lower and higher cell indices in the leaf node. The partitioning of an interior node is done by dividing the boundaries of the cell into equal sized buckets and calculate the cost function for bucket edges similar to BVH's cost function. The edge with the minimum cost function is chosen as the split position. The coordinate axis with maximum variation of centroids points is chosen as split axis. While the cost function for cell-tree construction is the same as the BVH, it can be shown that the formula can be simplified to the following form:

$$c(A,B) = t_{trav} + L_{max}N_A t_{incell} - R_{min}N_B t_{incell}$$

Similar to the BVH, cell-tree is first constructed in tree representation for debug simplicity. It is converted to compact linear representation afterwards to increase the traversing performance by making the cell-tree structure memory and cache friendly. In the linear representation, each node is followed by its left and right subtrees. The tree is traversed using a stack-based algorithm which starts from the root node. The sample point is tested against the bounding box of flow field's cells to test if it is inside the flow field bounds. During tree traversal, the sample point is tested against the left child's maximum extent, and right child's minimum extent in split axis direction, and the proper traversal path is chosen. If the sampling point is in the overlapping area of both left and right child, one of them should be traversed and the other should be kept in the stack in case no cells containing the sample point is found. A wise choice for next traversing node is the child which its split axis has greater distance to the sampling point. Similar to the BVH traversing, when the child index is negative, the sample point resides in a leaf node, and it should be tested against cells associated with the leaf node. The traversal pseudocode is shown in listing 3.9.

```
// celltree: the cell−tree constructed for an unstructured simulation grid
// s: sampling point
// returns the cell contatining the sampling point
Cell* findSurroundingCell(Celltree* celltree, Point* s) {
  int nodeNum = 0;
  Stack stack;
  // test if the sample point is inside the bounds of celltree.
  if(!inside(celltree−>bounds, s)) return nullptr;
  while(true){
    // if the current node is an interior node
```

```
    if(nodeNum >= 0){
      const LinearCellTreeInteriorNode* node = &celltree->m_interior_nodes[nodeNum];
      const bool insideLeftChild = s->getCoord(node->splitAxis) <= node->Lmax ;
      const bool insideRightChilld = node->Rmin <= s->getCoord(node->splitAxis);
      if(insideLeftChild && insideRightChilld){
        const float d0 = abs(s->getCoord(node->splitAxis) - node->Lmax);
        const float d1 = abs(s->getCoord(node->splitAxis) - node->Rmin);
        if(d0 >= d1){
          stack.push(node->children[1]);
          nodeNum = node->children[0];
        } else {
          stack.push(node->children[0]);
          nodeNum = node->children[1];
        }
      } else if(insideLeftChild && !insideRightChilld){
        nodeNum = node->children[0];
      } else if(insideRightChilld && !insideLeftChild){
        nodeNum = node->children[1];
      } else {
        if(stack.empty() == 0) break;
        nodeNum = stack.pop();
      }
    // current node is a leaf node
    } else {
      const LinearCellTreeLeafNode* node = &celltree->m_leaf_nodes[-nodeNum - 1];
      for(int i = node->lowIdx; i < node->highIdx; i++)
        if( insideCell( i, glm::vec3( point[0], point[1], point[2] ) ) ) return &cells[cell_index];
      if(stack.empty()) break;
      nodeNum = stack.pop();
    }
  }
  return nullptr;
}
```

Listing 3.9: **Cell-tree traversal pseudocode**. This procedure is used to locate the cell containing the sample point. Initially, The sample point is tested against the bounding box of the flow. If the point is in the flow field, it is tested against the boundaries of the child nodes in the split axis coordinate to specify the next cell in traversal path. When the sample point is located in a leaf node, it will be tested against its containing cells. During traversal, when the point is found in both of children, one of them is used for traversing, and the other is kept in a stack for traversing afterwards if a containing cell is not found. It is a wise decision to choose the child which the sample point is farther than its splitting boundary.

3.6. Cell-Walking

In previous sections of this chapter, constructing and traversing acceleration structure used for looking up the cell containing the sample point is discussed. During integrating a streamline, the acceleration structure should be traversed from the root every step which can be costly. It is possible to provide information about neighboring cells in the unstructured simulation grid and choose the step size small enough, to make the integration end up in one of the neighbors. In this case, looking up the next cell containing the sample point will be of $\mathcal{O}(1)$. Although this method also demands an acceleration structure for locating the cell containing the seeding point or in case where the sampling point is not in any of the neighboring cells, it can reduce the memory usage on devices like GPUs, and peripheral CPUs; therefore, the acceleration structure is required to be stored on the main CPU, and used for locate the cell whenever using the neighboring information is not possible.

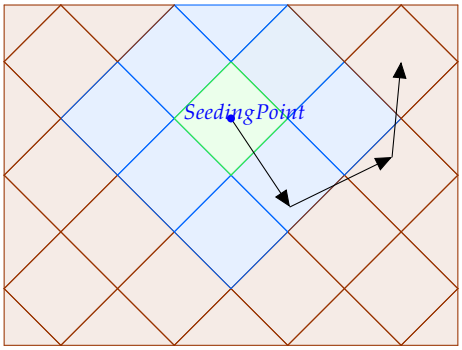


Figure 3.6.: **Cell-Walking**. The green cell containing the given seeding point. The cells colored as blue are the neighbors of the green cell. To advance the particle through this flow field, checking the neighbor cell is enough. The appropriate step size for this method should be small enough, to keep the next point inside one of the neighbors after integration.

Locating a cell in unstructured simulation grid using this method demands storing list of adjacent neighbors for every cell of the flow field which is called *adjacency cell information* array. This list can be extracted by iterating through points of every cell of flow field, and store index of cells which include the point. The next point The cell which contains the sample point is attained using an acceleration structure. In this master's thesis implementation, a grid is used for cases where the acceleration structure is demanding for cell lookup. By looking up the first cell containing the seeding point, the point can be advanced through the flow field just by testing the sampling point

Accel. St.	Mem. Limit	Construction	Traversal
Grid	$N_{Cells}.N_{Voxels}$	Simple	None
BVH	$(N_{Cells}-1).N_{Interior}+N_{Cells}.N_{Leafs}$	Quality depends on $N_{Buckets}$	Stack
Kd-tree	$N_{Leafs}.N_{Cells}+N_{Interior}$	Best possible Kd-tree	Stackless
Cell-tree	$(N_{Cells}-1).N_{Interior}+N_{Cells}.N_{Leafs}$	Quality depends on $N_{Buckets}$	Stack

Table 3.1.: **Acceleration structure Overview**. The memory limit, construction algorithm, and traversal algorithms of the four different acceleration structures are compared briefly.

inside the neighboring cells. The requirements of this method is to use very small step size to look up the next point inside one of the neighbors otherwise it is looked up using acceleration structure which is costly.

While this method does not require an acceleration structure during the streamline integration, its adjacency cell information array by itself has memory overhead. Additionally, its small step size constraint makes the integration slow for dense unstructured simulation grids,e.g., in wind tunnel dataset, an streamline of length 1.0 is achieved by step size 0.01, and 100 steps using a grid acceleration structure for cell look up, while the same streamline is achieved by step size 0.0001, and 10000 steps using cell walking.

3.7. Discussion

Table 3.1 shows a brief comparison between the memory limitation, construction method, and traversal method of described acceleration structures. As it is shown, grid and Kd-tree acceleration structures memory limits are tremendously high. For the grid, the highest memory limit is the evidence of a flow field which the whole cells are overlapping with the voxels. It can be fixed by increasing the number of voxels of the grid acceleration structure. The Kd-tree highest memory limit happens when the cost of splitting the root node is greater than traversing the cells of the flow field; in this case, all the cells of the flow field is always tested against the sample point which its cost can be approximated with $N_{Leafs}.N_{Cells}+N_{Interior}$. As the BVH and cell-tree are using the object partitioning approach, the objects are not duplicated in the acceleration structure, so the memory limit for them are acceptably low as shown in the table.

As it is discussed, the construction of grid acceleration structure is pretty straightforward, fast, and simple. While efficient BVH and cell-tree are constructed in 2 steps with a bottom-up approach. Both structure are using bucketing technique for partitioning the cells. While it usually split the cells very well but the number of buckets used for

partitioning has a very important role. The construction algorithm used for Kd-tree is fast and ends up with best possible Kd-tree.

Although the space partitioning algorithms suffer from memory limit, traversing them is faster. The grid does not need a traversal at all. The voxel containing the sample point is calculated by a simple mathematical operation. As the sample point is either in left or right child of a Kd-tree's interior node, the maximum number of traversing steps for a Kd-tree is also equal to height of the tree. As it is possible to have overlapping children's bounding boxes for both BVH and cell-tree structures, using stack during traversing the tree is necessary. The stack is used to keep track of traversal paths which are not tried. Therefore, the wise decision on which interior node to chose for traversal in next step plays an important role. Additionally, the stack stores memory during runtime and may reduce the performance while the traversing the tree is done on GPUs.

4. Streamsurface Generator (SSGEN)

The heart of this master thesis is the engine used for heterogeneous streamsurface computation and multi-GPU rendering. It is capable of utilizing multiple CPUs and GPUs for streamsurface computation, and multiple GPUs for rendering, simultaneously. It helps scaling the streamsurface computation and rendering in real-time by just adding new CPUs and GPUs to the computer system. In this chapter, the overall architecture of the application, and how the streamsurface generator engine fits inside is described. The implementation of streamsurface computation and rendering pipeline is presented in detail. In order to optimize the streamsurface computation, a proposed method for applying ray-packing and ray-sorting techniques in streamline computation is explained. In addition, a method similar to [**Schirski:50085**] is presented which eliminates the acceleration structure demands and used the neighboring information of the cells during streamline computation. Furthermore, different rendering methods implemented in the streamsurface rendering engine is described and the algorithm used to gather and combine the results of different GPUs in a final image is explained.

4.1. Software Architecture

The application's architecture is designed to utilize the scalability and portability features of heterogeneous computing. The goal is to achieve high performance streamsurface computation, and rendering. If GPUs are only used for streamsurface computation, the outcome triangles should be transfered to the main GPU for rendering. It is a high overload both for bandwidth and rendering which will drop the performance. The solution is to make each GPU responsible for rendering the triangles, it has computed on its own framebuffer and then gather all GPUs framebuffer's textures on the main GPU, and combine them together. Transferring the textures containing the result is much lightweight than streamsurface triangles. Additionally, compositing the results textures is less expensive than drawing the triangles directly, on the master GPU. As it is discussed in section 2.2, using all GPUs for rendering requires one OpenGL context for each one. Additionally, using OpenGL-CL interoperations demands creating one OpenCL context per device and define the rendering OpenGL context during OpenCL context creation. The OpenGL does not provide any procedures to specify the GPU hosting the OpenGL context; unless, a NVIDIA Geforce Quadro GPU is used. It is the

operating system that determines the which GPU hosts the OpenGL context. Determining the GPU hosting the OpenGL context is a bit tricky. In Ubuntu[**helmke2015ubuntu**] operating system, the OpenGL context creation on a desired GPU, a display should be connected to the desired GPU while the Xinerama[**ferguson1995x**] is disabled (deactivating features which makes moving a window from one display to other one possible). Therefore, the operating system will force the window's OpenGL context on the GPU which the its connected display containing the window. Figure 4.1 represents the device setup configuration used in this master thesis.

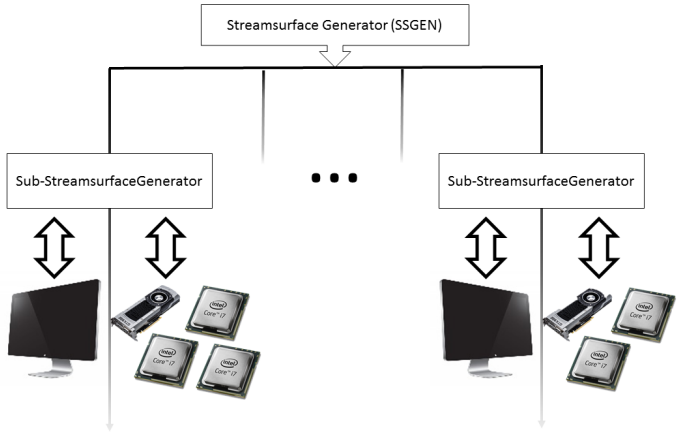


Figure 4.1.: **Device setup configuration** used in this master thesis to be able to use the multi GPU rendering.

To support the scalability of heterogeneous computing, the application is divided into a few sub-applications which can operate independently. For each connected display, a sub-application is allocated. Thereafter, the master application distribute the computation and rendering devices onto the sub applications. Each sub-application must contain at least one GPU, otherwise it will be removed. Finally, the CPUs are distributed among sub-applications. In this architecture, every sub-application has its own computation, and rendering manager which fires up its computations and drawing tasks share and schedule them. The sub-application's computation manager is responsible for scheduling task execution on the assigned computation resources. It determines the amount of the tasks based on device's current computation power weight factor's value. The computation power weight is a factor determines the device's computation power relative to a specific reference device. The Intel Pentium E860 which benefits from dual 3.0 GHz cores is chosen as reference device. Table 4.1 shows initial computation power weights for a few CPUs and GPUs. Meanwhile streamsurface

computation, each device's computation time is measured. The computation manager uses the timings to update the computation weights to establish a balance among the assigned computation devices timings. It improves the streamsurface computation performance. Additionally, the master application has a scheduler which is responsible for splitting the seeding points on the seeding line among sub-applications relative to each sub-application's overall computation power. The sub-application's overall computation power can be calculated by summing up the computation power values of sub-application's assigned devices. Figure 4.2 shows a general overview of the application architecture.

As the sub-applications are running as independent threads which compute and renders the streamsurfaces, synchronizing the threads are necessary. Listing 4.1 shows the model used to synchronize sub-application's threads. The sub-application's thread is always waiting for a task signal from the master StreamsurfaceGenerator thread. For example, updating the the sub-applications is done by storing the required data in a shared memory between the master thread and sub-applications. By sending a signal to sub-applications, the data about seeding plane, streamsurface tracing parameters, etc. is taken from the shared memory and stored in the thread's local memory. The master thread will wait for all the sub-applications to finish their updating and computing the new streamsurfaces. Waiting mechanisms can be implemented using mutex[**2009operating**], and condition variables[**Schling:2011:BCL:2049814**]. The Appendix **??** shows how this scenario can be implemented using the Boost[**Schling:2011:BCL:2049814**] C++ library. The same synchronization model is also used for initializing, drawing the sub-applications.

```
void master_thread(){
    while(true){
        // feeding the shared buffers used during the task.
        feed_share_buffer();

        // signal all the sub−applications to do the task.
        signal_all_subapps_do_task();

        // waiting for all the subapps to return done signal
        wait_for_all_subapp_done_singlas();
    }
}

void subapp_thread(){
    while(true){
        // waits for the incoming task from the master thread
        wait_for_task();
```

Device Name	Computation Power Weight (CPW')
NVIDIA GeForce GTX 980	8.0
NVIDIA GeForce GTX 970	6.0
Intel Core i7-4960X	4.0
TAHITI (AMD Radeon R9 280X)	6.0
PITCAIRN (AMD Radeon HD 7850)	4.5
Intel Pentium G860	1.0

Table 4.1.: **Initially assigned computation power weight** for a few number of devices.

```
        // handle the incoming task
        handle_task();

        // signal the master thread
        signal_task_done();
    }
}
```

Listing 4.1: **Synchronization** between the master and sub-application threads.

4.1.1. SSGEN Initialization

The application needs to do an initialization step to detect the connected devices, and create the required contexts for them. In this step, all the connected displays are detected. Operating system API calls (EnumDisplayMonitors API call in windows and XOpenDisplay API call in linux) can be used for detecting the connected displays. An instance of sub-application is allocated for each display. Each sub-application live through a separate independent thread. The sub-application initializes itself in the first step in the living thread, by creating a window and an OpenGL context for drawing in it. In the next step, the computation and rendering managers should be initialized. The computation initialization is done by detecting all the computation devices (CPUs and GPUs) and creating an OpenCL context for them. As the returned OpenGL renderer name, and OpenCL device name are not always similar (Specially for AMD GPUs), detecting the right OpenGL-OpenCL context match is done by measuring the connection speed between them. Corresponding to every OpenGL context, an OpenCL context is created on each GPU. There is an effort to keep the matching OpenCL context for an OpenGL context, and delete the others. This context can be detected by measuring the data transfer speed for copying an OpenGL buffer into another one using an OpenCL kernel. Listing 4.2 shows the kernel used for measuring the transfer speed. It simply reads the data from memory and writes it in another

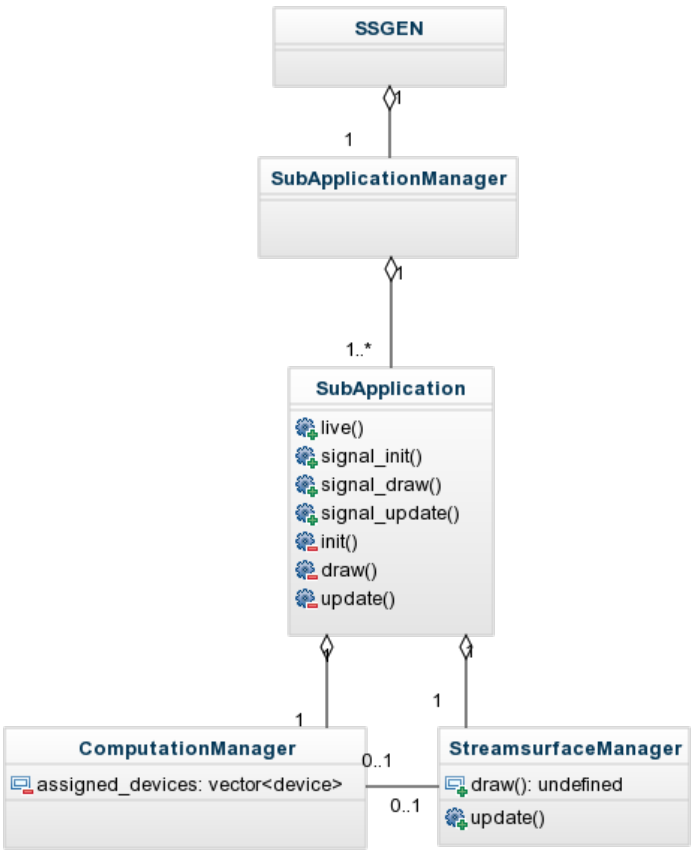


Figure 4.2.: **SSGen archicture**. In the left figure, it is shown that the application is divided into a few sub applications. Each sub application is independent of each other. Each takes a copy of the dataset, and part of the seeding point, and start generating the streamsurface from those points. The results from different sub applications are composited in the application's main window. The right figure shows a general overview of the sub application. Each one has its own computation and rendering manager. It manages computing and rendering streamsurfaces.

chunk. The required time for copying an specific amount of data from one buffer to other one is used to measure the transfer speed (GBps). The OpenCL context with maximum transfer speed is kept and the others will be deleted. Listing 4.3 shows the connection speed between different computation devices and the current active

renderer. In the first part the GeForce GTX 980 is the active renderer device, and in second part the GeForce GTX 970 is the activated. While the CPU has almost the same connection speed with both of them, There is a significant difference between data transfer speed when both contexts are on the same device compared to case where they reside on different ones.

```
__kernel void testTransferSpeed(
__global int* a,
__global int* b,
int num_threads) {
int thread_idx = get_global_id(0);

if(thread_idx >= num_threads) return;

b[thread_idx] = a[thread_idx];
}
```

Listing 4.2: **Transfer Speed Measurment** kernel.

```
#####################################################################
## Renderer: GeForce GTX 980/PCIe/SSE2
#####################################################################
##GeForceGTX9800
## Renderer Connection Speed: 93.668 GB / sec
##GeForceGTX9700
## Renderer Connection Speed: 0.913519 GB / sec
#####################################################################
##Intel(R)Core(TM)i7-4960XCPU@3.60GHzU @ 3.60GHz
## Renderer Connection Speed: 0.554983 GB / sec
#####################################################################


#####################################################################
## Renderer: GeForce GTX 970/PCIe/SSE2
#####################################################################
##GeForceGTX9800
## Renderer Connection Speed: 1.9351 GB / sec
##GeForceGTX9700
## Renderer Connection Speed: 159.134 GB / sec
#####################################################################
##Intel(R)Core(TM)i7-4960XCPU@3.60GHzU @ 3.60GHz
## Renderer Connection Speed: 0.863949 GB / sec
#####################################################################
```

Listing 4.3: **Comparison between the bandwidth speed** between different OpenGL contexts and OpenCL contexts.

As a result, each sub-application's OpenCL GPU context is declared. As, the CPUs does not have rendering capability, the OpenCL-OpenGL interoperations cannot be used for them. Therefore, the data should be copied to OpenGL buffer manually. Hence, the CPUs are distributed on the sub-applications uniformly.

4.2. Streamsurface Tracer

As described in chapter 2, the streamsurfaces are practically approximated by triangulating the streamlines starting from points on a seeding curve. In this section, the techniques used for computing the accurate streamlines, streamsurface refinement, and streamsurface triangulation in real-time are described. Additionally, some optimization techniques which can make the streamline/streamsurface computation faster are described.

4.2.1. Streamline tracing pipeline

In this master thesis, the seeding curves are defined on a seeding plane which its orientation and position can be modified. The seeding curves usually are straight lines. The equal distant seeding points on the straight line can be calculated simply by interpolating along it, e.g. if the arbitrary line from A to B is going to be divided into n segments, point i ($i = 0, 1, ..., n+1$) resides at $(i/(n+1) * B + (1 - i/(n+1)) * A$. In addition to the straight line, circle and bezier curve is implemented in this master thesis. To divide the circle into n equal distant segments, points $P = (\cos(i * 2\pi/n), \sin(i * \pi/n), 0.0)$ in the seeding plane coordinate system are transformed into world coordinate. For the bezeir curves, De Casteljau's algorithm can be employed for choosing the points on the curve. Therefore, to divide the curve into n segments, the points p_i is achieved by setting $t = i/(n+1)$ in Casteljau's algorithm. Although the points are not equal distant, the refinement strategy of the streamsurface computation which will be described in next section compensates it if the points should be chosen closer.

Streamline tracer takes one of the seeding points as input, and integrate it along the vectors stored in the vector field. As described in chapter 2, there are different integration methods. The Euler integration method is fast but inaccurate. In this master thesis, the Cash-Karp integration method with step adaptation is used which is described in chapter 2. The listing 4.4 shows how the point integration with Cash-Karp method in the vector field is implemented. The bt_ck_A and bt_ck_B are the constant values extracted from Cash-Karp's butcher tabelu described in chapter 2. The stepsize variable determines the length of the integration step. The error value for the current integration step can be estimated. If the estimated error value is less than minimum tolerable error value, the step size can be increased to have faster integration in the

vector field. Otherwise, if the error value is bigger than the maximum error value, the integration with current stepsize is not tolerable. The current integration step is marked as inaccurate by setting *StepAdapted* to -1. The integration directions are achieved by calling the derivate function in this listing.

```
float4 q[8];
q[0] = (*stepsize) * derivate(dataset, accelstruct, p0, lastCellIdx);
q[1] = (*stepsize) * derivate(dataset, accelstruct, p0 + bt_ck_A[1][1] * q[0],
        lastCellIdx);
q[2] = (*stepsize) * derivate(dataset, accelstruct, p0 + bt_ck_A[2][1] * q[0]
        + bt_ck_A[2][2] * q[1], lastCellIdx);
q[3] = (*stepsize) * derivate(dataset, accelstruct, p0 + bt_ck_A[3][1] * q[0]
        + bt_ck_A[3][2] * q[1] + bt_ck_A[3][3] * q[2], lastCellIdx);
q[4] = (*stepsize) * derivate(dataset, accelstruct, p0 + bt_ck_A[4][1] * q[0]
        + bt_ck_A[4][2] * q[1] + bt_ck_A[4][3] * q[2] + bt_ck_A[4][4] * q[3], lastCellIdx);
q[5] = (*stepsize) * derivate(dataset, accelstruct, p0 + bt_ck_A[5][1] * q[0]
        + bt_ck_A[5][2] * q[1] + bt_ck_A[5][3] * q[2] + bt_ck_A[5][4] * q[3]
        + bt_ck_A[5][5] * q[4], lastCellIdx);

// solution for the integration
q[6] = bt_ck_B[1][0] * q[0] + bt_ck_B[1][2] * q[2] + bt_ck_B[1][3] * q[3]
        + bt_ck_B[1][4] * q[4] + bt_ck_B[1][5] * q[5];

// error estimate
q[7] = bt_ck_B[0][0] * q[0] + /* skip bt_ck_B[0][1] */ +bt_ck_B[0][2] * q[2]
        + bt_ck_B[0][3] * q[3] + /* skip bt_ck_B[0][4] */ +bt_ck_B[0][5] * q[5];

// integration step
float4 p1 = p0 + q[6];

const float epsilon = length(q[6] - q[7]);

if (epsilon <= MIN_ERROR)
{
    (*stepsize) *= 2.0f;
    (*stepsizeAdapted) = +1;
}
else if (epsilon >= MAX_ERROR)
{
    (*stepsize) *= 0.5f;
    (*stepsizeAdapted) = -1;
}
```

Listing 4.4: **Cash-Karp integration** method.

The derivate function in listing 4.4 is looking up the cell containing the current point, it interpolate the values on the corners of the cell, and return the direction in that point. The body of derivate function is described in the listing 4.5. It uses the caching

technique, to make the streamline integration faster. As, it is plausible that the current point has not left the cell containing the previous point, first the previously cached cell is test against the current point. If the containing cell is not found, the acceleration structure is used to look up the vector field to find the containing cell. As described in chapter 2, testing if a point is inside a cell is done via calculating the barycentric coordinates and testing if they are between zero and one. As barycentric points can be reused in the interpolation step, they are output in this step. If a containing cell is found, the vectors in the corners of the cell will be interpolated to calculate the affective vector in the point. The different cell's interpolation methods are described in chapter 2. If no cells containing the sample point, a zero vector is returned.

```
// It is very likely to find the next point in the same cell
if ( (*lastCellIdx) > 0 &&
      inside(dataset->cell_box[((__global Grid*)accel_struct)->element_cells[(*lastCellIdx)]], point)){
      cell_idx = ((__global Grid*)accel_struct)->element_cells[(*lastCellIdx)];
 return dataset->cell_vector[cell_idx];
}

// Traverse the acceleration structure to find the containing cell.
cell_idx = findCellContainingPoint(dataset, accelStruct, point, barycoords);

// If a cell containing the point is found
if(cell_idx > 0)
    return interpolateDerivateInCell(dataset, cell_idx, point, barycoords);

// If no cells containing the point is found, return zero vector.
return zero_vec;
```

Listing 4.5: **Derivate look-up** function.

Sometimes, the integral curve cannot complete their path and they terminate suddenly. It usually happens when the integration step size is too large that the integral curve leaves the dataset or mesh boundaries. It is called *early termination*. To avoid it, when the sample point leaves the boundary or the derivate vector length became zero, the current point is thrown away and the previous point is being integrated with a smaller step(one-third of previous step size). Additionally, if the current integration is marked as inaccurate, the next point will be thrown away; the integration will be done with smaller steps in next iteration. If the integrated point is acceptable, it will be added to the streamline points. Listing 4.6 shows the body of streamline integration code.

```
// Trace the stream line
int i = 0;
int specialBackSteps = params->traceSpecialBackStepsLimit;
while(i < MAX_TRACE_STEPS){
```

```
        // Avoid early termination
        if(!inside( ( ( __global UnstructDataset* )dataset )->world_bounds, utilVars.current_point) ){
            if (i > 0 && specialBackSteps > 0){
            utilVars.stepsize *= 1.0f / 3.0f;
                utilVars.current_point = utilVars.previous_point;
            specialBackSteps--;
                continue;
            }
            else break;
        }

        utilVars.current_derivate = derivate(dataset, accelstruct, utilVars.current_point, &last_cell_idx);

        // Avoid early termination
        if (length(utilVars.current_derivate) == 0){
                if (i > 0 && specialBackSteps > 0){
            utilVars.stepsize *= 1.0f / 3.0f;
                utilVars.current_point = utilVars.previous_point;
                        specialBackSteps--;
                        continue;
                }
                else break;
        }

        utilVars.next_point =
            step_forward(dataset, accelstruct, utilVars.current_point,utilVars.current_derivate,
                &utilVars.stepsize, &utilVars.stepsize_adapted, &last_cell_idx);

        // stepsize must be decreased
        if (utilVars.stepsize_adapted == -1){
            specialBackSteps--;
            continue;
        }

        // add the current calculated vertex to the streamline vertices buffer.
        ...

        // ping-pong the previous-current-next point
        utilVars.previous_point = utilVars.current_point;
        utilVars.current_point = utilVars.next_point;
        i++;
}
```

Listing 4.6: **Streamline integration** kernel.

4.2.2. Streamsurface tracing pipeline

As mentioned previously, the streamsurface approximation practically is done by connecting successive integral curves (streamlines). One of the most important problems in streamsurface approximation is the refinement strategy. The strategy used in this thesis is similar to [CampPaper]. Two important constants used in this method are: D_{refine} which is the maximum possible distance between two adjacent streamlines to achieve the accurate streamsurface. If the distance between two adjacent streamlines is bigger than D_{refine}, the streamsurface is encountered as inaccurate. $D_{discontinue}$ which is the minimum acceptable distance between two adjacent seeding points. It is used for stopping the refinement algorithm in points where the surface is tearing. In this case, the seeding points are close enough to each other but the streamlines are still diverging far from each other due to discontinuity in the streamsurface. The following algorithm formulate how the refinement is done in this master thesis:

1. Initially, seeding points($c(i)$ on seeding curve C) are traced in the vector field to compute the initial streamlines on the seeding curve ($S_{c(i)}, i = 0, ..., N$).

2. For every two adjacent streamlines, the maximum distance is calculated and examined. If it exceeds the D_{refine}, the streamsurface does not have appropriate accuracy. To eliminate the inaccuracy, a new seeding point is placed in between these two streamlines' origins. It is assumed that the seeding curve is piecewise linear; therefore, the new seeding point is chosen the point in center of the origins.

3. If two seeding points' distance is less than $D_{discontinue}$, possibly there is a discontinuity in the streamsurface, therefore the refinement in that part of the streamsurface should be stopped. If the distance between, the new seeding point and adjacent streamlines origins is greater than $D_{discontinue}$, the new seeding point is inserted into a list called S_{insert}.

4. If the S_{insert} is empty, the refinement algorithm will be terminated.

5. The seeding points stored in S_{insert} are traced in the vector field, and the new integral curves are merged with the previous integral curve list. To guarantee the required accuracy, the distance between streamlines should be examined. Therefore, the algorithm continues step 2.

A big enough buffer ($5 * num\_init\_seeding\_points * max\_number\_of\_steps$) is used to store the streamline vertices. The required buffer for vertices of a streamline will be allocated statically by considering every maximum number of steps vertices as a streamline vertices. By using this strategy, some of the memory blocks of streamlines

which terminate before doing maximum number of integration steps are wasted. But it keeps the streamline vertices sequentially, therefore the coalescing will work perfectly on the GPU. Additionally, dynamic allocating memory from the whole buffer demands using an atomic counter to keep the number of allocated vertices which can drop the performance. The Triangles of the streamsurface are defined by indices to the vertices buffer. The buffer used to store the triangles is defined big enough ($5 * 2 * 3 * num\_init\_seeding\_points * max\_number\_of\_steps$). The memory used to keep the triangles made by vertices of a streamline is allocated statically from the big buffer. All of the triangles which has an edge on a streamline are kept successively. During the refinement and triangulation step, the adjacent streamlines should be declared. The order of the streamlines are defined via a doubly linked list in the OpenCL code. Two big enough arrays are defined to keep the previous and next streamline offset in the vertices buffer. Inserting a new streamline between two adjacent streamline is done by changing the indices in the doubly linked list arrays. The listing 4.7 shows how the new streamlines are allocated, and inserted between two diverged streamlines.

```
// if this line needs refinment.
if (needs_refinment){

  // new_line index
  int new_line = atomic_inc(num_seeds);

  if (new_line >= MAX_NUM_LINES){
    atomic_dec(num_seeds);
    return;
  }

  seeds[new_line] =
    (vertices[streamlines[stream0_offset + 1]].pos + vertices[streamlines[stream1_offset + 1]].pos) / 2.0f;

  // setting the new_line prev and next
  prev_lines[new_line] = ribbon_idx;
  next_lines[new_line] = next_lines[ribbon_idx];

  // The new_line is added in between the current line and next one
  prev_lines[next_lines[ribbon_idx]] = new_line;
  next_lines[ribbon_idx] = new_line;
}
```

Listing 4.7: **Doubly linked-list** used for storing the streamlines in appropirate order.

The listing **??** describes the strategy used for trianguling the adjacent streamlines, and compute normals of the vertices. This block of code, generates the triangles in one side of streamline which is basically half of a ribbon. The algorithm initialized by connecting the streamlines' starting points. This line is called active edge.The active edge moves

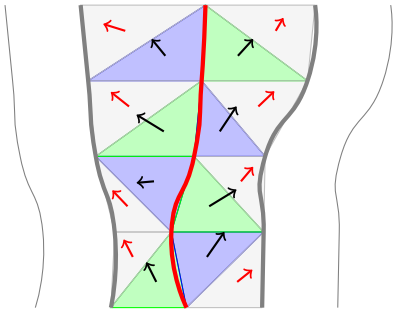


Figure 4.3.: **Triangulating the streamsurface**. While the streamsurface traingulation is done, the normals on the vertices are calculated by distributing the triangle normal vector on the points on the corner of the triangle. The parallelization on ribbons demands a barrier/mutex to avoid possible race condition for distributing the normals on the face's corner points. The parallelization is done on streamline. Therefore, the colored triangles (blue, and green) are stored for each streamline. The gray triangles (ones with red normal vector)are not added to the triangle list of the current streamline. Their normal is distributed on the streamline vertices which are part of them. Finally, the normal vector on each vertex should be normalized.

while the ribbon is being triangulated. The length of two diagonals connecting each active edge point and next point in the adjacent integral curve are examined. The shorter one is taken as the next active edge, and a triangle made of points on previous active edge and current one (one point is in common) are generated. The normal of this triangle can be calculated by taking the cross product of edges of triangle. This procedure terminates if the active edge move beyond any of the integral curve points. This algorithm is parallelized on the streamlines,i.e. every GPU thread will work on one streamline. Although, doing parallelization per ribbon makes it possible to represent the ribbon with a triangle strip of indices, the normal computations for common vertices among two ribbons may end up with race condition. In this case, some barrier/mutex mechanism is required to avoid changing a vertex's normal in two or more threads simultaneously. By changing the parallelization approach to streamlines, each streamline is generating the half of the triangles to ribbons it belong. The other triangles of the ribbon are also necessary for normal computation. In this Approach, the normals can be computed without using any barrier/mutex objects because the threads are not modifying any shared memories. Figure 4.3 shows how the triangles are calculated for a streamline, and the normal vector is calculated for each triangle.

In the refinement step, the maximum distance of the two adjacent streamline line is

computed by calculating the maximum active edge length.

```
int i = 0, j = 0;
while (i < MAX_TRACE_STEPS && j < MAX_TRACE_STEPS){
        // if one of the streamlines points are done
        if (i >= minLen || j >= minLen) break;

        // left diagonal, and right diagonal connection length
        left_diag = length(v_i_1 − v_j_0);
        right_diag = length(v_j_1 − v_i_0);
        if (left_diag < right_diag){
                normal = cross(v_i_0 − v_j_0, v_i_1 − v_j_0);
                vertices[s_j_0].normal += fabs(normal);
                i++;

                s_i_0 = s_i_1;
                s_i_1 = streamlines[stream0_offset + i + 1];
                v_i_0 = v_i_1;
                v_i_1 = vertices[s_i_1].pos;
        } else {
            add_triangle(s_j_1, s_j_0, s_i_0);
            triangle_indx++;

            normal = cross(v_i_0 − v_j_0, v_j_1 − v_j_0);
            vertices[s_j_0].normal += fabs(normal);
            vertices[s_j_1].normal += fabs(normal);
                j++;

                s_j_0 = s_j_1;
                s_j_1 = streamlines[stream1_offset + j + 1];
                v_j_0 = v_j_1;
                v_j_1 = vertices[s_j_1].pos;
        }
}
```

Listing 4.8: **Triangulation strategy** for adjacent streamlines.

4.3. Additional optimizations

While one of the main features of the OpenCL kernels is code portability, there are some circumstances where the code can be optimized for a specific architecture due to differences in the architectures, e.g. using the local memory on the CPU does not improve the performance because the local and global memories map on the same memory space on the CPU, but GPUs can benefit using the local memory because it is mapped to the on-chip memory on the GPUs, as mentioned in chapter 2, which is 100

times faster than uncached global memory[**cook2013cuda**]. Therefore, copying large chunks of data which is regularly used by the threads of workgroups that fits in the on-chip memory can have significant impact on the overall task execution performance. In such cases, it is beneficial to neglect code portability and write two versions of code one specific to the GPU architecture, and the other one specific to the CPU architecture.

In this section, a proposed method is presented to investigate the impact of packing the streamlines which their front points are spatially close, and advancing front in similar directions together in a package (the same workgroup which execute on the same SIMD). This method is pretty similar to the ray-packing and ray-sorting which is used in high performance graphics algorithms [**Eisenacher:2013:SDS:2600890.2600910**]. During the streamline integration, the acceleration structure (e.g. BVH-Tree or Cell-Tree) is regularly accessed by the threads doing the streamline integration on the GPU. Additionally, as Runge-Kutta demands looking up the derivate vector more than Euler, which will increase the number of acceleration structure accesses. It is of high probability that the sample points which has similar spatial locality and similar integration directions uses the same sub-tree during cell-lookup. Therefore if these sample points are packed together and the sub-tree which they use during traversal copied in the on-chip memory, the integration can speed up significantly. To pack coherent threads computing the streamlines together in a workspace, the front point of the stream and the derivate in those points are examined. First, the streamlines are divided into six cardinal direction bins based on their maximum dimension of the direction vector. The streamlines in a bin are first sorted along bin's maximum dimension. Then, every 4096 subsequent streamlines make a group together. The streamlines in a group are sorted based on derivate on the front point of streamlines. The streamlines in a group are packed into workgroups of 64 streamlines and traced together on the same SIMD. These streamlines will use the same sub-tree of the acceleration structure during integration. Therefore, copying the sub-tree into the on-chip memory can have significant impact on the streamline tracing performance.

Listing 4.9 shows the algorithm used to detect the sub-tree which all the front points of a workgroup are residing inside. The sub-tree should be the biggest possible sub-tree which fits into the GPU's on-chip memory. As the sampling points of a packet should resides in the same sub-tree, finding the sub-tree which should be copied to the on-chip memory can be done only by one of the threads in the workgroup and the others can wait for it. To make finding the sub-tree easier, two new fields are added to the interior nodes of a tree: sub-tree size, and the parent index (Listing 4.10). Finding the sub-tree fits in the on-chip memory is done by locating the leaf node containing the front point of the first streamline. The sub-tree well fits to the on-chip memory can be found by traversing the tree upward and testing if the number of interior nodes in the sub-tree is greater than the maximum number of nodes can be stored in the on-chip memory. The

sub-tree which its parent size has a size greater than on-chip memory size, is the best possible sub-tree for copying into the on-chip memory.

```
__local local_BVHTree subtree;

// The first thread is going to find the subtree which is going to be copied in the local memory.
if(thread_local_idx == 0){

  subtree_interior_idx = -1;

  for(int i = 0; i < local_dim; i++){
        subtree_interior_idx = locateLeafIdx((__global UnstructDataset*)dataset,
                                             (__global Tree*)accelstruct, seeds[thread_idx + i]);
        if(subtree_interior_idx >= 0) break;
  }

  if(subtree_interior_idx >= 0){
        // Go up the tree to find maximum subtree which can reside in the local memory.
        subtree_interior_idx =
            ((__global Tree*)accelstruct)->leaf_nodes_info[ subtree_interior_idx ].parent;
        int next_subtree_interior_idx = subtree_interior_idx;
        __global LinearInteriorNodeInfo* interior_nodes_info =
          &(((__global Tree*)accelstruct)->interior_nodes_info[ subtree_interior_idx ]) ;
        while(next_subtree_interior_idx > 0 &&
          getSubTreeSize(interior_nodes_info->last_interior, next_subtree_interior_idx) < ON_CHIP_MEM_SIZE){
          subtree_interior_idx = next_subtree_interior_idx;
              next_subtree_interior_idx = interior_nodes_info->parent;
              interior_nodes_info =
                &(((__global Tree*)accelstruct)->interior_nodes_info[ next_subtree_interior_idx ]);
        }
  }
}

subtree.interior_nodes = local_interior_nodes;
subtree.leaf_nodes = local_leaf_nodes;

// Wait all the threads to find the subtree containing the
barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 4.9: **Detecting the appropirate sub-tree** which seeding points in a packet reside in and are small enough to be copied into GPU's local memory.

```
struct LinearBVHInteriorNodeGPU: public LinearBVHInteriorNode {
int parent; //!< index of parent node
int last_interiror; //!< index of the last interior node in sub-tree
};
```

Listing 4.10: **GPU's linear BVHTree's interior nodes** structure definition..

The found sub-tree should be copied directly to the on-chip memory and used for tracing streamlines. Listing 4.11 presents the procedure of copying the sub-tree from the global memory to the on-chip memory. To avoid changing the indices of the leaf nodes and interior nodes, the base index for interior and leaf nodes are kept. The *async_work_group_copy* is used to copy the interior and leaf nodes into the local memory. This instruction copies the data from the global memory to the local one using all the work items of the workgroup implicitly. A *wait_group_events* is used to wait on finishing the copy task by all the threads.

```
//! Point to the node of the tree which will be copied to the local subtree with its childs.
__global LinearBVHInteriorNode* subtree_globalptr =
  &(((__global Tree*)accelstruct)->interior_nodes[subtree_interior_idx]);

int last_subtree_interior_idx = subtree_info_globalptr->last_interior;
int n_subtree_interior_nodes = last_subtree_interior_idx - subtree_interior_idx + 1;

if(thread_local_idx == 0){
  subtree.interior_base_idx = subtree_info_globalptr[1].parent;
  subtree.leaf_base_idx = subtree_info_globalptr[0].leaf_lowidx;
  subtree.tree = accelstruct;
}

// Event used for waiting on the copying the data from global to local memory.
event_t copy_event;

// Copy the data for interior nodes from the global memory to local memory
copy_event =
  async_work_group_copy((__local long2*)(&subtree.interior_nodes[ interior_nodes_beg_idx ]),
    (__global long2 *)(&subtree_globalptr[ interior_nodes_beg_idx ]),
    n_subtree_interior_nodes * sizeof(LinearBVHInteriorNodeGPU), 0);

int leafnodes_lowidx = subtree_info_globalptr->leaf_lowidx;
int leafnodes_highidx = subtree_info_globalptr->leaf_highidx;
int n_subtree_leaf_nodes = leafnodes_highidx - leafnodes_lowidx;
__global LinearBVHLeafNode* subtree_leaf_globalptr =
  &(((__global Tree*)accelstruct)->leaf_nodes[leafnodes_lowidx]);

// Copy the data for leaf nodes from the global memory to local memory
copy_event = async_work_group_copy((__local long*)&subtree.interior_nodes[leaf_nodes_beg_idx],
  (__global long*)&subtree_globalptr[ leaf_nodes_beg_idx ],
  n_subtree_interior_nodes * sizeof(LinearBVHLeafNode), copy_event);

// Wait all the threads to finish the copying task.
wait_group_events(copy_event)
```

Listing 4.11: **Copying the appropirate sub-tree** into the on-chip memory.

To locate the cell containing the front point of a streamline, first the traversal algorithm should be done on the local copy of the sub-tree. If the cell was not found in the local sub-tree, the traversal should be started from the acceleration structure root. The sub-tree which is already look up will not be traversed again. As the threads in a workgroup may diverge from each other, they should be re-arranged after an specific predefined number of steps. In this master thesis, this algorithm is not implemented completely. The main goal was to study the possible improvement in streamline by applying the ray-packing from high performance graphics field in scientific visualization field. The investigation results can be found in chapter 5.

4.4. Rendering Stream Surface

As described in previous section, streamsurfaces are made of ribbons. The streamsurfaces vertices are stored in a single buffer. Each ribbon is defined by indices of face vertices which shape the final representation of the ribbon. To draw the streamsurface via OpenGL's draw calls, one draw call should be done for each ribbon. It can ends up with huge number of draw calls which will drop the rendering performance. OpenGL provides a glMultiDrawElementsIndirect API function which allows providing the draw call parameters through a buffer. The structure of each buffer's element is shown in listing 4.12.

```
typedef struct {
    uint count;
    uint instanceCount;
    uint firstIndex;
    uint baseVertex;
    uint baseInstance;
} DrawElementsIndirectCommand;
```

Listing 4.12: **The screen space ambient occlusion (SSAO)** post-processing stage.

Each element contains information about number of indices to be drawn, number of instances, first index in the draw call, base vertex index, and base instance index. To use this draw call, a buffer of DrawElementsIndirectCommand should be filled and bound to GL_DRAW_INDIRECT_BUFFER. Then the glMultiDrawElementsIndirect must be called with number of indirect commands in the bounded buffer of indirect calls. Listing 4.13 shows how this draw api function should be called.

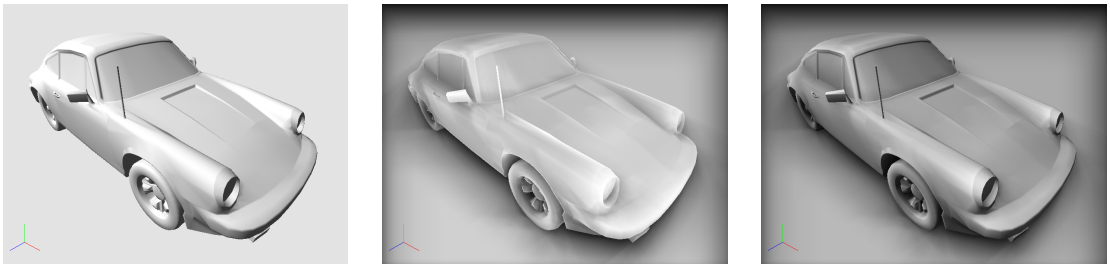
```
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, indirect_buffer);
glMultiDrawElementsIndirect(GL_TRIANGLES, GL_UNSIGNED_INT, 0, 4, 0);
```

Listing 4.13: **The screen space ambient occlusion (SSAO)** post-processing stage.

The following structure is filled in the end of triangulation kernel. Correspondingly, it is a buffer shared among OpenGL and OpenCL contexts. The streamsurfaces in this master thesis can be rendered with three different techniques: Phong-illumination, Phong-illumination with screen-space ambient occlusion, and semi-transparent streamsurfaces. In the next two sections the underlying structure of these techniques is described.

4.4.1. Screen-Space Ambient Occlusion

Ambient occlusion is a term used to describe the amount of light reaching a point on a diffuse surface based on directly visible occluders. It usually used to add the approximated global illumination effect to the final rendered image. The ambient occlusion makes the depth, curvatures, and spatial proximity's perception more clear. The main idea of ambient occlusion is to trace rays through the normal-oriented hemisphere, and use the percentage of rays that do not hit any geometry in a specific distance as the ambient occlusion term.



(a) Phong Illumination (b) Ambient Occlusion (AO) (c) Phong Illumination with Ambient Occlusion

Figure 4.4.: **Screen Space Ambient Occlusion (SSAO)**. Ambient occlusion is an approximation of global illumination effect which is computed based on directly visible occluders surrounding a specific point. It makes the depth, curvature, and spatial proximity perception easier.

There are three different solutions for calculating the ambient occlusion: ray tracing ambient occlusion, object space, and screen space. Calculating the ambient occlusion by ray tracing is done by tracing rays through normal-oriented hemisphere, and counting the number of rays which hit in a specific radius. This procedure will need computation power significant computation power. Therefore, it is precomputed as an ambient occlusion texture for static object that is used during rendering. The ambient occlusion in object space can be computed by approximating the polygonal objects with surface

elements which possibly occlude each other. Each surface element's ambient occlusion factor can be calculated with respect to other surface elements. The ambient occlusion factors on the surface elements are used to achieve the ambient occlusion value on the original object's surface. The ambient occlusion with this method can be precomputed and used in runtime. The screen space ambient occlusion method can be computed in real-time. Therefore, it is well suited for dynamic scenes. It usually applies as a post-process fragment shader which uses the z-buffer texture values, and a surface normal texture to calculate the ambient occlusion.

In this master thesis, the screen-space ambient occlusion (SSAO) is implemented as the streamsurfaces are computed on the fly and the ambient occlusion cannot be precomputed. Additionally, applying the screen space ambient occlusion needs less computation power than ray tracing method than the other two methods. Furthermore, applying it can be done by adding a new pass to the rendering pipeline which is more compatible to the current streamsurface rendering method.

To compute the ambient occlusion in screen space by adding a post-processing stage, another texture is attached to the framebuffer to store the normal vector, and the z-component of every fragment position in view-space. This texture is filled during rendering the phong illumination representation of the object and is called heightmap. In the post-processing stage, heightmap is used to compute the ambient occlusion value. The algorithm is described in listing 4.14.

```
// Get texture position from gl_FragCoord
vec2 P = gl_FragCoord.xy / textureSize(sNormalDepth, 0);

// normal and depth
vec4 ND = textureLod(sNormalDepth, P, 0);
vec3 N = ND.xyz;
float my_depth = ND.w;

float occ = 0.0, float total = 0.0;

// For each random point (or direction)...
for (int i = 0; i < point_count; i++){

// Get direction
vec3 dir = points.pos[i].xyz;

// Put it into the correct hemisphere
if (dot(N, dir) < 0.0f) dir = −dir;

  // f is the distance we've stepped in the ray direction, z is the interpolated depth in ray direction.
  float f = 0.0f;
  float z = my_depth;
```

```
    // We're going to take 4 steps - we could make this configurable
    total += 16.0f;
    float r = ssao_radius / 16.0f;

    for (j = 0; j < 16; j++){
      // step toward viewer in dir direction
      f += r;
      z -= dir.z * r;

      // Read depth from current fragment
      float their_depth = textureLod(sNormalDepth, (P + dir.xy * f), 0).w;

      // If we're obscured, accumulate occlusion
      if ((z - their_depth) > 0.0){
      float d = length(f * dir.xy);
      occ += 1 / (d*d) * dot(N, dir);
      }
    }
}

// Calculate occlusion amount
float ao_amount = 1.0f - occ / total;

// ...
```

Listing 4.14: **The screen space ambient occlusion (SSAO)** post-processing stage.

Similar to [], the heightmap is used to approximate the scene in a 2D representation; the ambient occlusion is computed by marching rays through normal-oriented hemisphere in the image space (2D). For each fragment the depth value, and the normal vector in view space is read from the heightmap. From every fragment center, a few number of rays are traced in random directions through the hemisphere with a fixed step in a specific extend which is defined by the normal-oriented hemisphere's radius. If the ray has collision with the height map, the ray is considered as the occluded, and amount of occlusion is calculated. The ambient occlusion for a pixel is defined by the occlusion value divided by the number of traced rays multiplied by the number of tracing steps. In this master thesis, to get a better perception of depth, the goal is to make the surfaces occluding each other directly clearer. Therefore, the importance sampling is used to make directly occluding surfaces more clear. Accordingly, the random ray direction is generated from a cosine density function, and the occlusion value is computed by calculating the dot product of the normal and occluded ray direction. To make the occlusion in nearer pixels more clear, the $1/d_2$ is also multiplied by the occlusion value. For streamsurface rendering, the ray marching extend is chosen small to make the disjoint surfaces more clear. Figure 4.4 shows the ambient occlusion

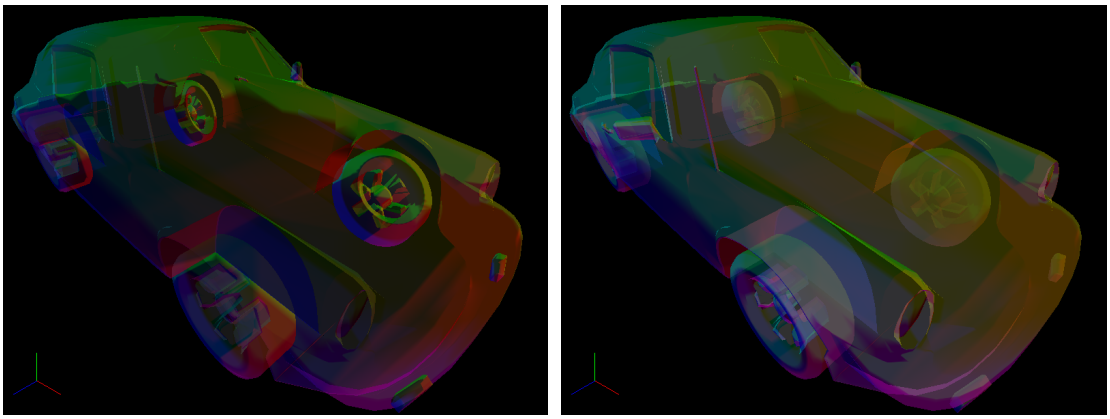
in action for rendering car model. Figure **??** shows the ambient occlusion applied to streamsurface rendering.

4.4.2. Semi-Transparent Surfaces

Rendering non-opaque objects is one of classical problems in computer graphics community. Rendering scenes containing semi-transparent objects demands drawing them in visibility order (front-to-back or back-to-front). Figure 4.5 represents the car model, which its components are rendered once in wrong, and one in correct order. As it is shown when the rendering is done in wrong order, the mirror is totally disappeared and the wheel behind the car's body seems like it is in front of it. One solution to rendering the opaque and non-opaque objects together correctly is to first sort all non-opaque triangles back-to-front order. Then, enable depth buffer testing and writing for rendering opaque objects. Finally, render the non-opaque triangles in visibility order while the depth test is enabled but the depth writing is disabled. The alpha-blending stage is utilized during drawing the non-opaque fragments to get the correct rendering result. While sorting the objects demands high computation power, it cannot always solve the problem. Figure 4.6 shows a case which sorting the triangles in front-to-back, or back-to-front oder is impossible. Therefore, calling the draw functions in order is not possible, and if the triangles were non-opaque the correct final result cannot be achieved.

Previously known method used to handle rendering semi-transparent objects is depth-peeling by doing n passes over the scene to extract n layers deep in it and then blend the passes to render the correct order of semi-transparent objects. This method requires huge rendering power. Furthermore, it is view-dependent and by changing the view-point or view-orientation, all the computations need to be done from scratch.

In computer graphics terminology, Order-Independent-Transparency (OIT) is meant to describe techniques used to correctly render non-opaque overlapping objects without sorting them explicitly before rendering them. There are different ways for implementing this technique e.g. using A-Buffer method[REF]. The technique implemented in this master thesis is very similar to [AMD]. In this technique, for each pixel on the screen a linked-list of fragments (fragment color, and its depth value) covering it are stored during rendering the scene. Afterwards, in a separate pass all linked-lists are sorted in depth-order, and then blended. To implement this method, a big enough buffer to keep all the fragments produced during scene rendering, a texture for storing the linked-lists' head pointers, and an atomic counter for keeping the next free fragment buffer element's index are required. Each time, a fragment is produced during scene rendering, one of fragment buffer's elements is allocated for it, and the atomic counter is increased. The head pointer for this pixel will change to the new fragment, and



(a) Draw calls in wrong order (b) Draw calls in correct order

Figure 4.5.: **Impact of ordering the objects on transparency**. In the left figure, the car's components are rendered in wrong order. As it is shown, the mirror is totally not visible, and the wheels behind the car's body seems like they are in front of it. The right figure, renders the car's components in the correct order.

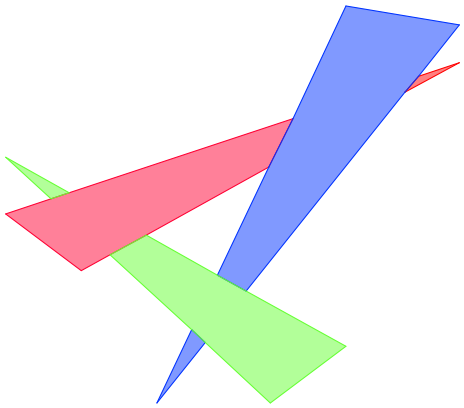


Figure 4.6.: **Impossible to sort**. Sorting these triangles in back-to-front or front-to-back order is impossible. Therefore, the draw functions cannot be called correctly to the right final result.

the old one is stored with this fragment. Additionally, the fragment's color, depth, its specular factor are stored to be used during sorting, and blending of the fragments. To keep, the buffer storing the fragments small, the OpenGL's API float packing/unpacking instructions are utilized, so previous element in the linked list, fragment's color, it's

depth, and it's specular factor can be stored in a 32-byte memory. To sort the linked lists per fragment's depth and blend their colors, a quad covering the whole screen is rendered in a separate pass. The fragment shader in this pass, unpacks and sorts the fragments based on their depth values with bubble sort algorithm[REF]. To make sorting faster, the linked-list is copied to a local array. The sorting and blending are applied on it. After sorting the array, the fragments colors are blended together, and the final color is written to the framebuffer. Listing XX shows how the fragments are sorted, and blended in the fragment shader.

```
// Maximum Number of fragments per
#define MAX_FRAGS 40

// Temporary array used for sorting fragments
uvec4 tmp_frag_list[MAX_FRAGS];

void main(void) {

    // Copy the linked list in local array
    uint curr_idx, frag_count = 0;
    curr_idx = imageLoad(headptr_image, ivec2(gl_FragCoord).xy).x;
        while (curr_idx != 0 && frag_count < MAX_FRAGS){
        uvec4 frag = imageLoad(frag_buffer, int(curr_idx));
        tmp_frag_list[frag_count] = frag;
        curr_idx = frag.x;
        frag_count++;
    }

    // Sort the fragments based on depth value
    uint i, j;
    if (frag_count > 1){
        for (i = 0; i < frag_count - 1; i++){
            for (j = i + 1; j < frag_count; j++){
                uvec4 fragment1 = tmp_frag_list[i];
                uvec4 fragment2 = tmp_frag_list[j];

                float depth1 = uintBitsToFloat(fragment1.z);
                float depth2 = uintBitsToFloat(fragment2.z);

                                    // TODO: Back-To-Front Order
                if (depth1 < depth2){
                    tmp_frag_list[i] = fragment2;
                    tmp_frag_list[j] = fragment1;
                }
            }
        }
    }
```

```
    // Blending the sorted fragments
    vec4 final_color = vec4(0.0f);
    for (i = 0; i < frag_count; i++){
        vec4 frag_color = unpackUnorm4x8(tmp_frag_list[i].y);
            vec4 frag_specularity = unpackUnorm4x8(tmp_frag_list[i].w);
        final_color = mix(final_color, frag_color,frag_color.a) + frag_specularity ;
    }
    color  = mix(color_background, final_color, final_color.a) ;
        gl_FragDepth = fragment_list[fragment_count − 1].z;
}
```

Listing 4.15: **The order independent transparency (OIT)** fragment sorting and blending stage.

4.4.3. Compositing

As it is described in previous sections, each sub-application renders on its frame buffer. To get a complete representation of the streamsurface, the framebuffers of the sub-applications should be composited. In the previous section, three different rendering modes are mentioned which are supported in this master thesis: Phong-illumination, Phong-illumination with screen-space ambient occlusion (SSAO), semi-transparent rendering. Composition of framebuffers for each rendering modes is specific. To composite the results of sub-applications, textures assigned to their framebuffers should be transfered to the main application's window OpenGL context. It is possible to get the contents of each texture on CPU and copy it to a texture in main window's OpenGL context but it works inefficient and slow. Additionally, sharing textures among two OpenGL contexts which resides on two different GPUs is not possible. NV_COPY_IMAGE's OpenGL extension provides an API function for copying a texture from one OpenGL context to another one. The prototype of the API function is shown in listing 4.16.

```
void glXCopyImageSubDataNV(Display *dpy,
        GLXContext srcCtx, uint srcName, enum srcTarget, int srcLevel,
        int srcX, int srcY, int srcZ,
        GLXContext dstCtx, uint dstName, enum dstTarget, int dstLevel,
        int dstX, int dstY, int dstZ,
        size width, sizei height, sizei depth);
```

Listing 4.16: **Copying texture from one OpenGL context to other one** can be done efficiently and fast by using glXCopyImageSubDataNV API extension function..

In Phong-illumination rendering mode, the result from different sub-applications can be composited easily, by rendering a quad fully covering the screen area, and doing

composition for each sub-application's result framebuffer in a separate pass. In each pass, the color of the sub-application's fragments and its depth value is written while the depth testing and writing are enabled. The depth testing stage, determines if this fragment will be overwritten on current content of the final framebuffer or not. The fragment shader for this stage can be found in listing 4.17 .

```
layout (binding = 0) uniform sampler2D color_texture;
layout (binding = 1) uniform sampler2D depth_texture;

out vec4 color;

void main(void) {
    color = texture(color_texture, fs_in.tex_coord);
    gl_FragDepth = texture(depth_texture, fs_in.tex_coord).x;
}
```

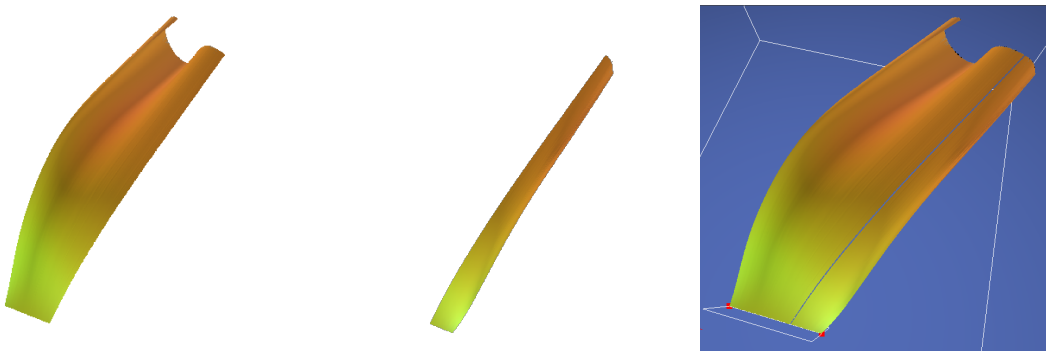
Listing 4.17: **Compositing Phong-Illumination textures** to get the complete representation of streamsurface.

To render the screen-space ambient occlusion, an additional texture should be written in the output called the heightmap. Compositing strategy is similar to Phong-illumination rendering mode: a quad covering the screen is rendered, and in the fragment shader the fragment color, depth, and height_normal value of a sub-application is considered to be written to the main framebuffer. They will be written to the framebuffer if it passes the depth test. This pass should be iterated for all sub-application's output textures. The listing 4.18 is the code used for compositing the result textures from different.

```
layout (binding = 0) uniform sampler2D color_texture;
layout (binding = 1) uniform sampler2D depth_texture;
layout (binding = 2) uniform sampler2D normal_height_texture;

layout (location = 0) out vec4 out_color;
layout (location = 1) out vec4 normal_height;

void main(void) {
    color = texture(color_texture, fs_in.tex_coord);
    gl_FragDepth = texture(depth_texture, fs_in.tex_coord).x;
    normal_height = texture(normal_height_texture, fs_in.tex_coord);
}
```

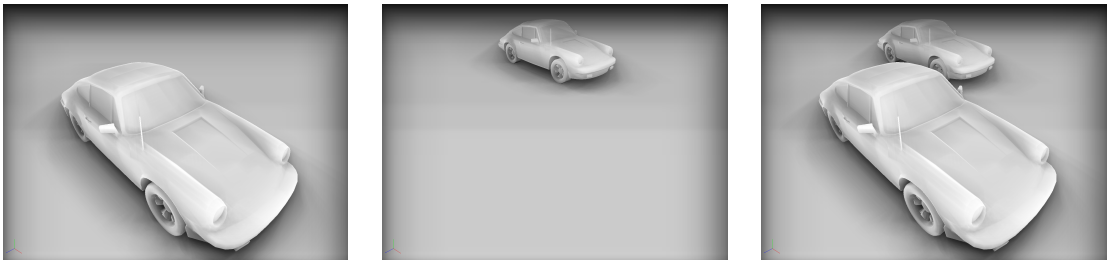
Listing 4.18: **Compositing Phong-Illumination with SSAO textures** to get the complete representation of streamsurface.

There are four different textures coming out of each sub-application during rendering the scene in the order independence transparency mode: color, depth, fragment buffer, and head index buffer. To composite these textures from different sub-applications,



(a) GPU 0 Result (b) GPU 1 Result (c) Composited Result

Figure 4.7.: **Compositing sub-application results**. The results rendered on two separate GPUs are shown in the left and middle figures. They are composited in the right image.



(a) GPU 0 Result (b) GPU 1 Result (c) Composited Result

Figure 4.8.: **Compositing sub-application results** with screen space ambient occlusion. The SSAO texture for results rendered on two separate GPUs are shown in the left and middle figures. Compositing the result rendered textures in a single texture results in SSAO shown in the right image.

multiple passes should be done for merging the fragment buffers in one fragment and head index buffer. In each pass, a sub-application texture results are merged into the main application's fragment buffer and the head index buffer is recomputed. As sorting should be done for each sub-application result and blending can be done once on the main fragment buffer, they are separated into two stages. Listing 4.19 shows how the fragment buffer, and head index buffer from a sub-application is merged into main application's ones in a single pass. First, the fragments of the sub-application's results are sorted based on depth in a local array. Then, the new fragments from closest to deepest are inserted in its place in the current fragment's linked list.

Finally, If some of the new fragments are behind the current fragments in the fragment buffer, they should be written to the fragment buffer and the head should change. As, imageAtomicExchange is a costly atomic function, there is an effort to call it once for the deepest new fragment and not for all. Therefore, the head of the list is written in a if after copying all the fragments to the fragment buffer. Additionally, the newly added fragment's linked-list needs to be chained to the previous linked-list. This is done by writing the old head value to the new linked-list first element's next fragment index. The result fragment buffer, and head pointer buffer for the main application is passed to a fragment shader for blending. Figure XX shows blending the results from different sub-applications is done via these procedures.

```
// Maximum Number of fragments per
#define MAX_FRAGS 40

// Temporary array used for sorting fragments
uvec4 tmp_frag_list[MAX_FRAGS];
uvec4 new_frag_list[MAX_FRAGS];

void main(void) {

// Copy the linked list in local array
uint curr_idx, frag_count = 0;
curr_idx = imageLoad(headptr_image, ivec2(gl_FragCoord).xy).x;
while (curr_idx != 0 && frag_count < MAX_FRAGS){
    uvec4 frag = imageLoad(frag_buffer, int(curr_idx));
    tmp_frag_list[frag_count] = frag;
    curr_idx = frag.x;
    frag_count++;
}

uint new_frag_count = 0;
uint headptr_idx = imageLoad(new_headptr_image, ivec2(gl_FragCoord).xy).x;
curr_idx = headptr_idx;
while (curr_idx != 0 && new_frag_count < MAX_FRAGS){
    uvec4 frag = imageLoad(new_frag_buffer, int(curr_idx));
    new_frag_list[new_frag_count] = frag;
    curr_idx = frag.x;
    new_frag_count++;
}

// Sort the new_frag_list with bubble sort.
...

// merging the new fragments into current sorted fragments.
uint i, j;
while(i < new_frag_count && j < frag_count){
```

```
    uvec4 new_fragment = new_frag_list[i];
    uvec4 fragment = tmp_frag_list[j];

    float new_depth = uintBitsToFloat(new_fragment.z);
    float depth = uintBitsToFloat(fragment.z);

    if(new_depth >= depth)
        j++;
    else
    {
        int curr_fragment_idx;
        uint new_frag_idx = atomicCounterIncrement(frag_counter);

        // not the last element in current sorted fragment buffer
        if(j < frag_count - 1)
        {
                        curr_fragment_idx = tmp_frag_list[j+1].x;
        }
        else
        {
            curr_fragment_idx = headptr_idx;
        }

        new_fragment.x  = fragment.x;
        fragment.x = new_frag_idx;
        imageStore(frag_buffer, int(curr_frag_idx), fragment);
        imageStore(frag_buffer, int(new_frag_idx), new_fragment);
        i++;
    }
}

uint new_frag_idx, prev_frag_idx, link_frag_idx;
for(int i_prime = i; i_prime < new_frag_count; i_prime++){
    uvec4 new_fragment = new_frag_list[i_prime];
    uint new_frag_idx = atomicCounterIncrement(frag_counter);
      if(i_prime > i)
      {
          new_fragment.x = prev_frag_count;
        imageStore(frag_buffer, int(new_frag_idx), new_fragment);
    }
    else
        link_frag_idx = new_frag_idx;
}

// If the new fragments are appended to the fragment buffer list, head pointer should changed.
if(i < new_frag_count){
    int old_head = imageAtomicExchange(headptr_image, ivec2(gl_FragCoord.xy), uint(new_frag_idx));
    new_frag_list[i].x = old_head;
```

```
    imageStore(frag_buffer, int(new_frag_idx), new_frag_list[i]);
}
```

Listing 4.19: **The order independent transparency (OIT)** fragment composition and sorting. This fragment should be executed for each sub-application result textures.

In addition to the streamsurface, the final representation has the graphic user interface (GUI), bounding box and outer surface of the dataset, and ground floor. The GUI is implemented with AntTweakBar[REF]. To get an overall representation of the dataset's bounds, the bounding box of the dataset is rendered on the screen. The user can also see the outer surface of the dataset's mesh. The outer surface of the dataset is extracted during loading the mesh from file by processing every face, and its neighboring cells; for each cell's face neighboring cells can be looked up. If it is empty, this face is part of the outer surface. The VTK's VTU and OpenFOAM reader strongly support these features. Finally, the application draws a ground floor on the screen with a spot light effect rendered on it to make representation of the streamsurface more clear.

5. Results and Discussions

In chapter 3, different acceleration structures for speeding up finding the cell containing the front point of a streamline in integration were presented. Chapter 4 proposed a heterogeneous computing solution to compute streamlines/streamsurfaces on all CPUs and GPUs of a computer system to improve the streamsurface resolution while keeping the performance. In the following chapter, these methods are evaluated and a short discussion of achieved results are presented. The experiments use two main sample scenes:

- **Wind Tunnel** is a simulation mesh made of around 4 million tetrahedral cells. The size of the dataset with velocity result is approximately 180 MB.
- **Car** is a simulation mesh made of approximately 20 million tetrahedral cells. The size of the dataset with velocity result is approximately 1.7 GB.

5.1. Peripheral Tools

Testing proposed algorithms and acceleration structures and evaluating them is of high importance. Two peripheral tools which were developed to simplify streamline/streamsurface computation test: The first one is the function to simulation mesh converter. It is mainly developed to generate simulation meshes that the correctness of the result streamline/streamsurface can be perceived easily. It generates a simulation mesh using function $f(x, y, z)$ values on the mesh's points as velocity vectors. If the derivate function $F'(x, y, z)$ is passed as the function to this tool, the streamlines/streaamsurfaces generated in this simulation mesh should look like $F(x, y, z)$ surface representation. The correctness of a function's surface representation is easily verifiable. The second tool is implemented to increase the density of the cells in a dataset to do some experiments on performance of different acceleration structures in case where the meshes get high-resolution.

5.1.1. Function to Grid Converter

Certainly, testing the result of streamline/streamsurface computation algorithms is of high important. In this thesis, a simple application is implemented which gets

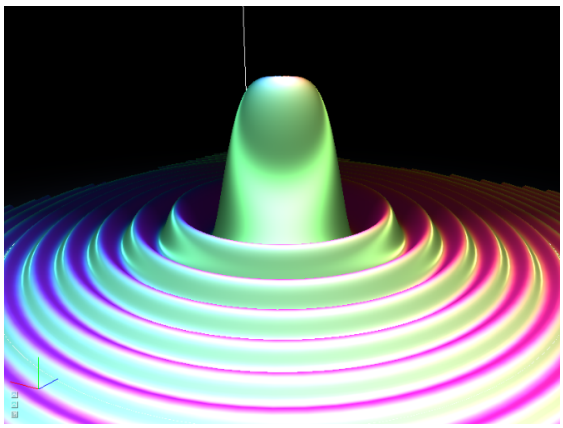


Figure 5.1.: **Function surface reconstruction** done using the streamsurface computation and rendering modules. The mesh is generated by the function to grid converter using the tangent vectors toward the coordinate system center on the surface of $F(x,y,z) = \frac{sin(f(x,z))}{f(x,z)} - y, f(x,z) = a\pi\left(x^2+z^2\right)$.

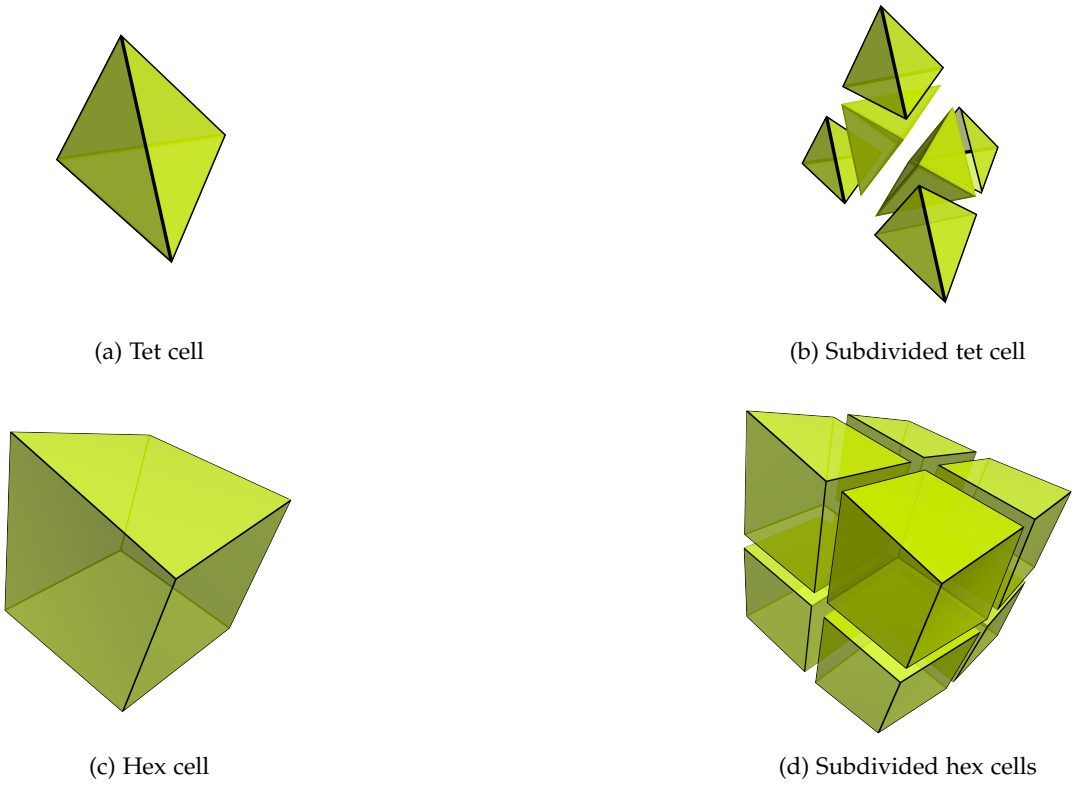
the derivate function $f'(x,y,z)$ as input and generates a grid of hexagons storing tangent vectors on the corners of the hexagons as velocity result values. The streamsurface/streamlines computed and rendered in the generated simulation mesh should look the same as the function surface representation in mathematics. As an example, the $F(x,y,z) = \frac{sin(f(x,z))}{f(x,z)} - y, f(x,z) = a\pi\left(x^2+z^2\right)$ surface is reconstructed by seeding points on a seeding circle in the generated simulation mesh. The velocity vector $\vec{D}$ on mesh points are generated by first calculating the gradient vector ($\vec{nabla}$) on the surface at the streamline's front point which is the normal vector on the surface:

$$\overrightarrow{\nabla} = \begin{bmatrix} \frac{\partial f(x,z)}{\partial x}\left(\frac{f(x,z)\cos(f(x,z)) - \sin(f(x,z))}{(f(x,z))^2}\right) \\ -1 \\ \frac{\partial f(x,z)}{\partial z}\left(\frac{f(x,z)\cos(f(x,z)) - \sin(f(x,z))}{(f(x,z))^2}\right) \end{bmatrix}$$

The Calculated normal vector and the streamline's front point shape a plane which is tangent to function's surface representation. All the vectors in this plane are also tangent to the surface of the reconstructed function. One of them should be stored on simulation mesh point. The vector directing the streamlines/streamsurfaces toward the coordinates system's origin should be used during streamline/streamsurface integration. The cross product of the normal vector and the vector connecting this point to the coordinate

system's origin is calculated ($\vec{T}$). The cross product of the normal vector $\vec{N}$, and $\vec{T}$ results in vector $\vec{D}$ which is tangent to the function surface and is oriented toward the coordinate system's origin. Figure 5.1 shows the reconstructed function surface.

5.1.2. Mesh Subdivider



(a) Tet cell

(b) Subdivided tet cell

(c) Hex cell

(d) Subdivided hex cells

Figure 5.2.: **Cell subdivision** is used to increase the resolution of the simulation mesh. A tetrahedron is subdivided into four tetrahedrons and two pyramids. Additionally, a hexagon is subdivided into eight smaller hexagons.

Mesh subdivider is a tool used for increasing the resolution of the simulation mesh. It is mainly designed to study different acceleration structures when the simulation mesh's resolution increases. It divides each cell into smaller cells (different kinds of smaller cells) to generate the same dataset but with a larger amount of cells. The results at the new points are linearly interpolated between available points. Therefore, the streamline/streamsurface representation should remain the same. During the evalua-

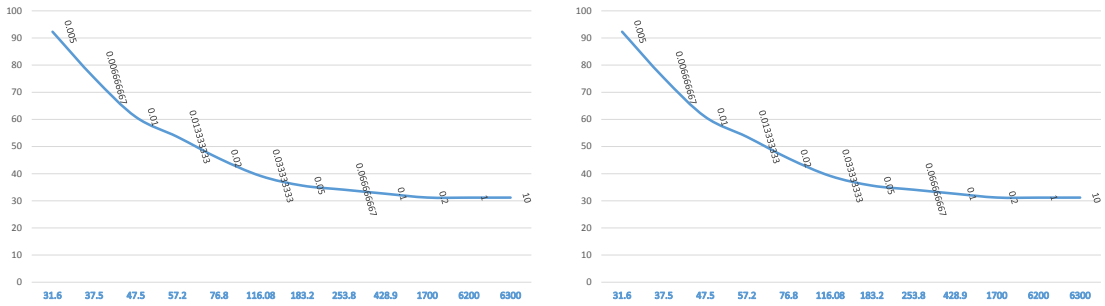
tion phase, it is expected that the streamline/streamsurface computation performance decreases with increasing subdivision of the original simulation mesh's cells. The goal is to investigate the impact of simulation mesh's resolution on the performance of different acceleration structures. As many of the available meshes are made of tetrahedrons and hexagons, subdivision methods are proposed for them: Tet cells are divided into four tetrahedrons, and two pyramids. Hexagon shape cells are subdivided into eight smaller hexagons. Figure 5.2 presents how the tetrahedrons and hexagons are subdivided into smaller cells. The set of resulting tetrahedrons, and hexagons keeps the original mesh's shape.

5.2. Results and Evaluations

In this section, the results of the main contributions of this master thesis are presented and their performance is evaluated. Initially, the investigations on different acceleration structures performance are presented. The performance are experimented The aim is to classify the dataset types w.r.t. acceleration structures and present which acceleration structure can provide the best performance for a specific dataset type. In the later subsection, the performance of the heterogeneous system is studied and the results are presented.

The test system has multiple components: two GPUs (an NVIDIA GeForce GTX 980, an NVIDIA GeForce GTX 970) and one CPU (Intel Core i7-4960X 3.6 GHz) with 32 GB RAM. The SSGen algorithms are implemented in two versions: single-threaded, and heterogeneous computing. The single-threaded version is mainly implemented for evaluating different accleration structures and streamline/streamsurface computation performance for high resolution simulation meshes. Furthermore, it is used to test the basics of algorithms used for computing streamlines and streamsurfaces. The heteregenous computing version is the version which applies the heteregenous computing and rendering context in algorithms for computation and rendering. It distributes computations and rendering tasks on available resources. The performance of streamlines/streamsurface computation and rendering is measured in this version.

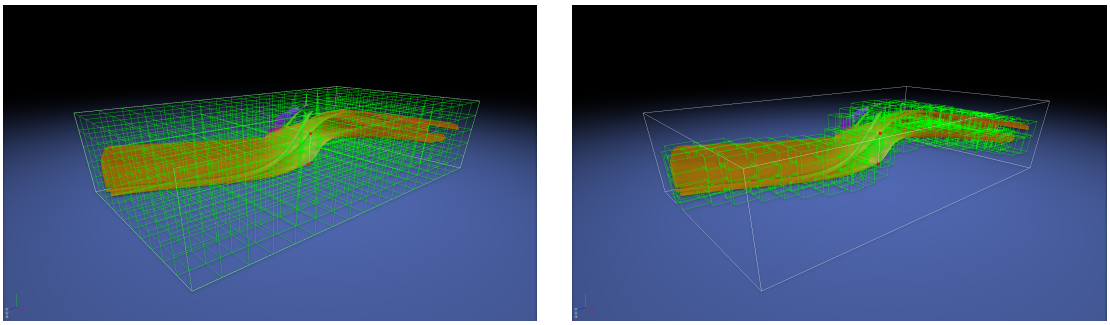
The acceleration structures are used to accelerate finding the cell containing the streamline's front point. Proper acceleration structures with appropriate properties can have significant impact on streamline computation performance. Two main factors used for acceleration structure comparison are: their size in the memory which is called *size*, number of checks needed to look up the cell containing the sample point's position which is called *checks per traverse*. In this master thesis, the performance of the grid, Kd-tree, BVH, and cell-tree are investigated. The first two are built based on a space partitioning approach and others have a cell/object partitioning approach. Each



(a) Wind tunnel dataset

(b) Car dataset

Figure 5.3.: **Kd-tree cost function factors adjustment**. The Kd-tree's behavior is determined by the ratio between t_{incell} and $t_{traverse}$, and maximum number of cells in the leaf node. Figure 5.3a presents the number of checks per traverse-size curve for the wind tunnel dataset, and figure 5.3b present them for the car dataset. The size of the Kd-tree is being adjusted by changing the ratio between incell and traversal times. The lables on the curve represents the ratio for every point. The expriements shows using a ratio between 0.05 and 0.1 provides the maximum performance.



(a) Grid

(b) BVH

Figure 5.4.: Grid and BVH built for the wind tunnel dataset.

acceleration structure has a few parameters which determine its behavior and efficiency. Regular grid's performance can be adjusted by the its resolution. By increasing the resolution of the grid, the number of cells corresponding to each grid's element will decrease. Consequently, the number of checks per traverse decreases that end up with grid's performance improvement. Performance-wise, the ideal acceleration structure for computing streamlines is a high-resolution grid which maps each cell to one element.

This acceleration structure usually use larger memory than the mesh itself. Therefore, the performance of acceleration structures and size in the memory should be considered together. BVH's behavior is controlled by adjusting the maximum number of nodes per leaf. The cell-tree's behavior is similarly controlled by adjusting the maximum number of cells per leaf node. Kd-tree's behavior is determined by adjusting the ratio between t_{incell} and $t_{traverse}$ in the cost function, and maximum number of cells per leaf node. To compare the acceleration structures, an appropriate ratio should be determined for the kd-tree. Then, the size of the kd-tree can be adjusted by setting the maximum number of nodes per leaf. Figure 5.3 shows the performance of the kd-tree when different ratios between the t_{incell} and $t_{traverse}$ are applied by keeping the maximum number of cells per leaf to 20. The horizontal axis is the size of the kd-tree in Megabytes (MB). The vertical axis is the number of checks per traverse for computing streamlines that define the streamsurface skeleton. It is concluded that a ratio between 0.05 and 0.1 provides the best Kd-treee's performance. As it is shown, the kd-tree's checks per traverse-size curve is an asymptotic curve. The tangent of the asymptotic curve which is the maximum performance of Kd-tree. The level of this line is adjusted by the maximum number of cells in the leaf node. Figure 5.4 shows the grid and BVH acceleration structures built for the windtunnel dataset.

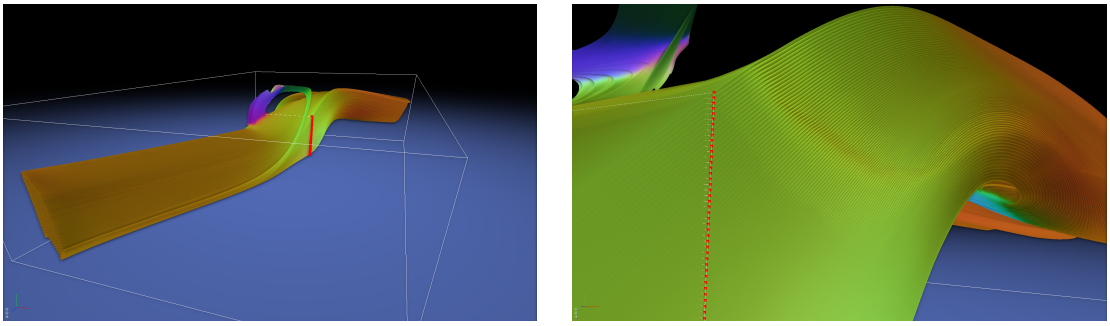
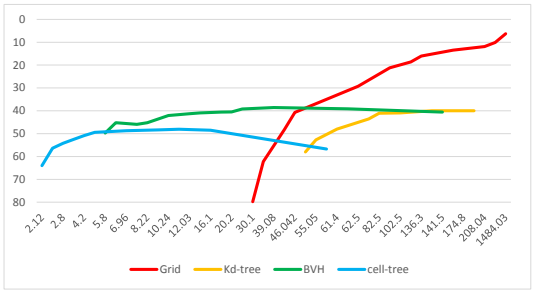
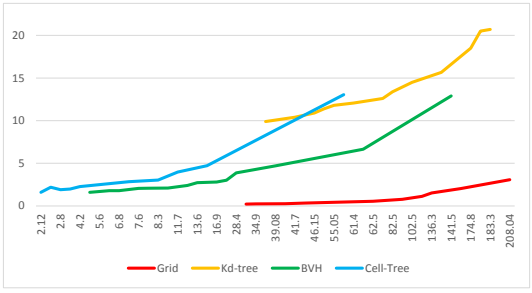


Figure 5.5.: **Streamsurface stack used for performance measurement**. Eighty streamsurfaces which each has 1000 starting seeding points and are integrated for 1000 steps. It is used to measure performance of different acceleration structures for streamlines of different kinds of streamsurfaces.

After determining the ratio between Kd-tree's incell and traversal values in the cost function, it is possible to compare the grid, Kd-tree, BVH, and cell-tree. Each acceleration structure has one parameter which determines its size in the memory. The goal is to determine the parameter values where each acceleration structure performs



(a) Performance (checks per traverse) - Acceleration structure size (MB)



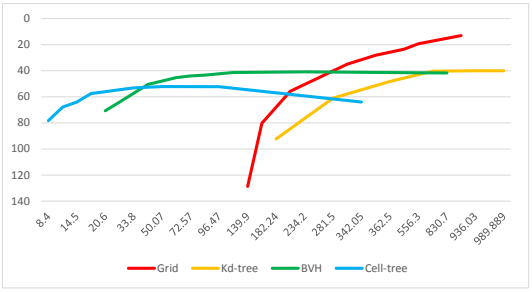
(b) Build Time (s) - Acceleration structure size (MB)

Figure 5.6.: **Acceleration structures performance and build time (no subdivision)**.

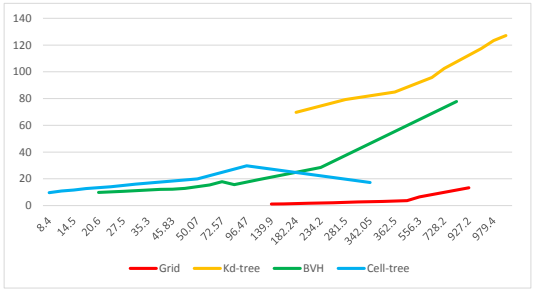
Acceleration Structure	Size (MB)	Avg. Perf. (checks per traverse)	Build Time (s)
Grid	136	16.05	1.53
Kd-tree	102.5	40.9	14.5
BVH	13.6	40.9	2.74
Cell-tree	6.96	48.7	2.85

Table 5.1.: **Optimum acceleration structures for the windtunnel**. While grid's performance is 2.5 times more than the other acceleration structures, its size is 10 times of BVH, and 20 times of cell-tree. Additionally, grid is very fast in building. Although BVH's performance is 20% above cell-tree, its size in memory is 2 times of cell-tree. BVH and cell-tree building algorithms are very similar, therefore their times are similar.

efficiently. Performance is compared in terms of the average number of checks to be done in every traversal for computing 80 streamsurfaces (each streamsurface made of 1000 streamlines that are originating from 1000 seeding points on a seeding curve which are integrated 1000 steps). The result streamsurfaces are shown in figure 5.5. While acceleration structures using cell-partitioning approach (BVH, cell-tree) reorders the dataset's cells list in a way that leaf nodes keep the beginning and end cell indices inside leaf nodes, instead of storing all the indices explicitly; the size of cell partitioning acceleration structures are less than space-partitioning ones (grid, and Kd-tree) which must keep indices of corresponding cells. The hierarchical approaches(BVH, cell-tree, and Kd-tree)'s minimum number of checks per traverse (maximum performance) are bounded by hierarchy's number of levels,i.e. to look up a given cell, the hierarchy should be traversed down to the leaf node which has cost of doing checks $\mathcal{O}(\log n)$

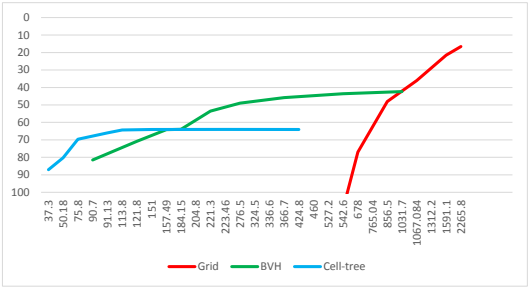


(a) Performance (checks per traverse) - Acceleration structure size (MB)

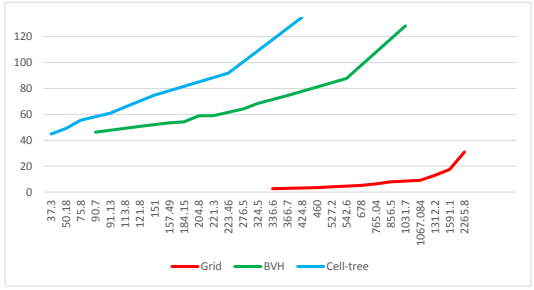


(b) Build Time (s) - Acceleration structure size(MB)

Figure 5.7.: **Acceleration structures performance and build time (1-subdivision)**.



(a) Performance (checks per traverse) - Acceleration structure size(MB)



(b) Build Time (s) - Acceleration structure size (MB)

Figure 5.8.: **Acceleration structures performance and build time (2-subdivision)**.

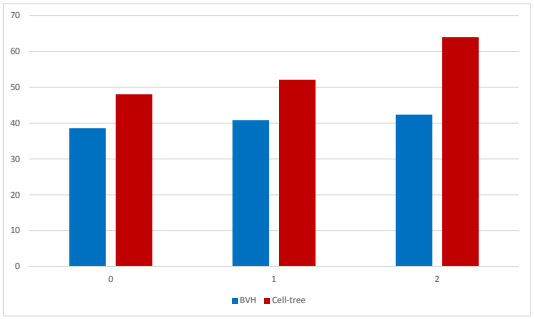
times, which is equal to the height of the hierarchy. In most cases locating the cell containing a given point demands back-tracking (using stack to go back in the hierarchy and traverse another sub-tree) which drops the performance far below this value.

Figure 5.6a presents the performance of acceleration structures of different sizes and different types. As can be seen, the grid's performance can be improved by increasing its resolution. However, increasing the resolution of the grid also increases memory demand. Kd-tree's maximum performance is very close to BVH's one while its size in the memory is 9 times more. BVH and cell-tree curves are asymptotic curves. Both of them have upper performance bound which is determined by the asymptotic line. The cell-tree reaches its optimum performance when its size in the memory is half of the BVH's size. This is due to the nature of cell-tree which has smaller leaf nodes in size than BVH. But keeping the bounding box around the cells instead of split point along

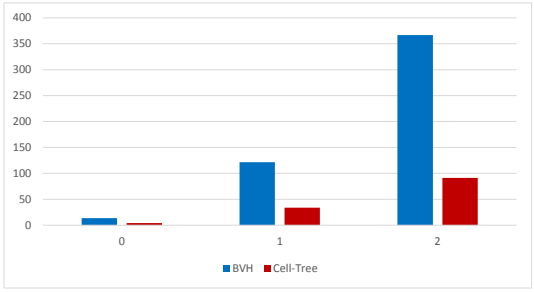
Subdivision Levels	Size (MB)	Number of Cells (approx.)
0	133.7	2.5 M
1	802.2	12.5 M
2	4144.7	54.1 M

Table 5.2.: **Wind tunnel subdivided meshes sizes**.

the split axis will decrease the number of checks to be done for a cell-lookup. Therefore, BVH's maximum performance is 25% more than cell-tree. Table 5.1 shows the build times for BVH, cell-tree, optimum Kd-tree, and a grid (136 MB, and 16.05 checks per traverse performance). The statistics shows that the grid building procedure is fast. Its performance can be improved until each cell is mapped to one grid's element. BVH and cell-tree has the same building procedure, therefore building times are close. Kd-tree's building is very slow compared to others. Figure 5.6b shows how the building times growth with size of acceleration structure.



(a) Grid and BVH performance statistics by increasing simulation mesh's resolution level



(b) Grid and BVH size statistics by increasing simulation mesh's resolution

Figure 5.9.: **Comparison between BVH and Cell-tree statistics**. Bars in figure 5.9a shows that the cell-tree's performance drops with increasing the resolution of the simulation mesh (25-50%). Figure 5.9b presents the size of optimum BVH grows faster than cell-tree by increasing the simulation mesh's resolution (BVH's size in the memory is between 2-8 times of the grid's size.)

Figure 5.7 and 5.8 show the performance of acceleration structures when the mesh subdivider tool is used to increase simulation mesh's resolution. Table 5.2 shows the size of wind tunnel's simulation mesh with different subdivision levels. As the mesh is made of tetrahedrons, the number cells are approximately multiplied by 6 (the pyramids are not subdivided). The goal is to evaluate acceleration structures performance when

the simulation mesh resolution increases. Figure 5.7a and 5.8a present the performance of different acceleration structures after subdividing the simulation mesh for one and two steps. Considering figure 5.6a, 5.7a, and 5.8a with increasing resolution of the simulation mesh, the grid's curve is being shifted to the right side of the diagram, i.e. growth of appropriate grid's size in memory is faster than BVH and cell-tree. Figure 5.8a shows that the optimum Kd-tree's performance is close to the BVH's one while it is using 6 times more memory. Furthermore, it is concluded that the BVH has better performance than cell-tree thanks to storing bounding box containing the inside cells instead of the boundary plane's position along split axis which cell-tree is storing. Additionally, increasing the resolution of the simulation mesh will reduces the cell-tree's size in memory but can drop the hierarchy performance up to 50%. Figure 5.9a, 5.9b compare optimum BVH and cell-tree performances and their size in memory. Although cell-tree's size in the memory grows slower than BVH, its performance drops faster. Figures 5.7b and 5.8b show that the grid's acceleration structure is very fast to build, and Kd-tree is very slow. Having a closer look, cell-tree and BVH with maximum performance building takes the same amount of time as their build algorithms are similar.

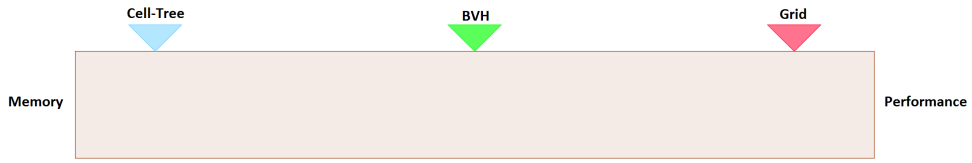
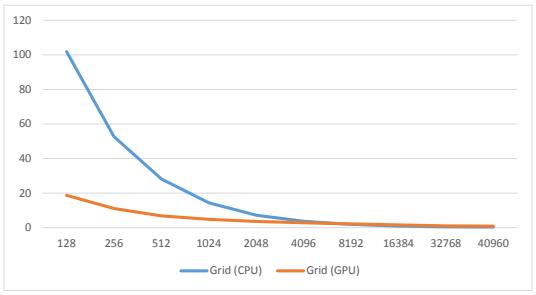


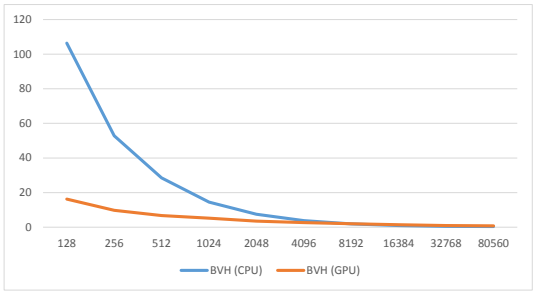
Figure 5.10.: **Acceleration structures classification**. This figure shows the conclusion of all acceleration structure evaluations. The cell-tree is memory-wise ideal, and the grid is performance-wise ideal. An acceleration structure which is a balanced among them is BVH. The acceleration structure of the choice is dependent on the use-case and available resources.

As a conclusion of the discussed results, the acceleration structure of choice is very dependent on the requirements and available resources. If enough memory is available and the performance is of main importance, grid is the best acceleration structure. In discussed examples, grid needs at least 25 percent of the simulation mesh size to provide similar performance to the BVH. Cell-tree is mainly a memory oriented acceleration structure. It provides an acceptable performance compared to its size in memory e.g. for all subdivided cases, cell-tree's maximum performance is achieved when its size is 4-5% of the mesh size in the memory. BVH provides 25-50% higher performance than cell-tree while its size is around 10% of the simulation mesh.

Although these acceleration structures are used in high performance graphics algorithms, the conclusions and acheived performances are significantly different. It basically originates from the nature of their application use-cases (ray-tracing or cell locating). The goal of ray-tracing is to find the nearest intersection point of the ray and objects in the scene. In contrast, streamline integration demands looking up the front point of a streamline. It causes the grid acceleration structure to become the ideal acceleration structure for cell-look up while it has a poor performance in ray-tracing (all corresponding objects in the grid's element that a ray passes should be checked for intersection).



(a) Grid

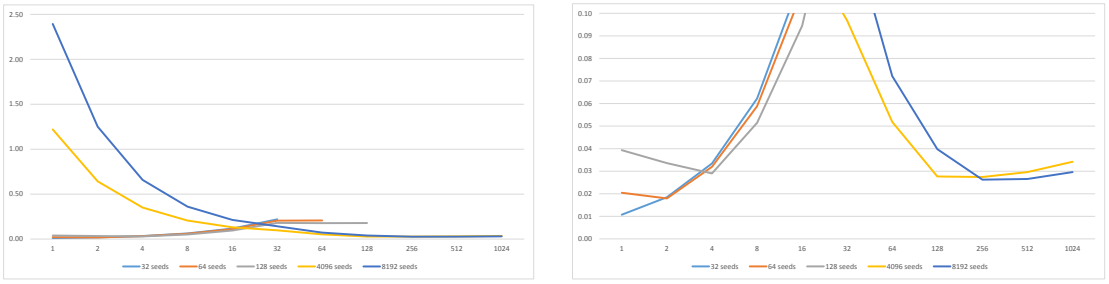


(b)

Figure 5.11.: **Grid and BVH performance on CPU and GPU**. Execution time of the streamline computation kernel utilizing grid and BVH acceleration structures verifies evaluated results. Additionally, the CPU is above the GPU curve for a smaller number of seeding points; because the GPU does not have enough tasks to exploit its computation power.

To verify the performance evaluations which is done, a similarly evaluated grid and BVH are compared in terms of streamline integration. Figure 5.11 shows the timings for computing streamlines of a streamsurface originating from a seeding curve consisting of a different number of points. The BVH size in the memory is 13.6 MB and the average required number of checks to find a given cell is 40.9. Similarly, the grid size in the memory is 46 MB and the average required number of checks to find a give cell is 40.7. Figure 5.11 shows the timings for computing different numbers of streamlines on CPU and GPU. As it was shown by evaluations, the curve achieved by using grid as acceleration structure on CPU/GPU is very similar to curve using BVH as acceleration structure on CPU/GPU. Additionally, considering the GPU and CPU curves, when a small number of seeding points is used, there are not enough tasks to do for GPU. Therefore, its computation power cannot be exploit and its performance is less than CPU for a smaller number of seeding points. As the number of seeding

points increases, enough number of tasks are being available for the GPU which cause the GPU outperform the CPU.

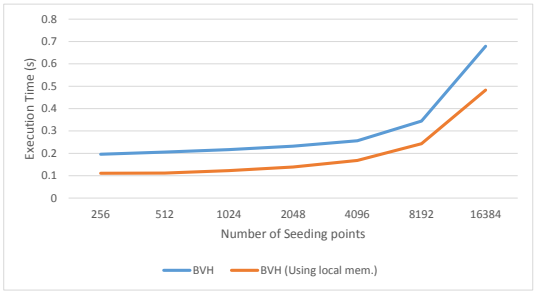


(a) (b)

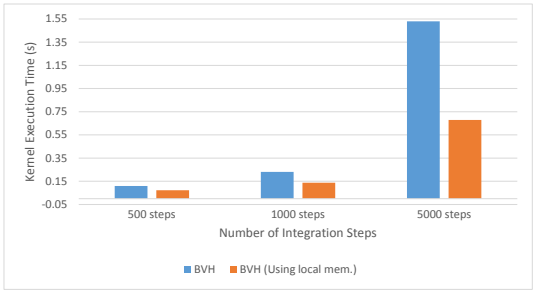
Figure 5.12.: **Streamline kernel execution time by workgroup size**. It is shown that the optimum number of group size is achieved when the threads computing the streamlines are divided into 32 workgroups. As the test is done on a GTX 980, the GPU has 16 SMMs. As the kernels has a lot of memory accesses and diverge in the code, it is more efficient to first distribute the kernel execution tasks on SMMs. To remove memory latency, it is efficient to execution two kernels per SMM. Therefore, it can be concluded the streamline computation threads should be divided into 32 different groups.

As described in chapter 2, kernel execution demands determining the appropriate workgroup size. Figure 5.12a shows the execution time of streamline integration kernel using different workgroup sizes. Having a close look at the execution time of the streamline computation (Figure 5.12b), it can be concluded that the minimum execution time (maximum performance) is achieved when the seeding points are divided into 32 workgroups (workgroup size is equal number of seeding points divided by 32 - It should be a power of 2 number. Otherwise, the performance will drop -). As there are many memory accesses in the streamline computation kernel and divergence in kernel execution on SMMs, it is more efficient to first distribute the threads on SMM computation units (GTX 980 has 16 SMM units). Additionally, mapping at least two streamline computation thread to one SMM can improve the performance by hiding the memory access latency.

In chapter 4, a streamline packing method was proposed to improve using BVH acceleration structure on the GPU by reducing the number of global memory accesses on the GPU. Figure 5.13 represents the maximum possible performance improvement if streamline packing is used. In this expriement, all the seeding points are one point in



(a)



(b)

Figure 5.13.: **Streamline-packing performance upper bound using BVH**. ???

the dataset space to make the wraps 100% coherent. Additionally, the step size is chosen to be small enough, so the streamlines do not leave the sub-tree's volume that is copied to the local memory. It shows that the maximum possible performance improvement for streamline integration using local memory is between can be acheived is 30-40%. Additionally, it is concluded that the local memory utilization can improve low number of seeding points; because when the number of seeding points are increased, ???. This method's performance is improved if the number of integration steps are increased.

5.2.1. Result Representation

Water is missing.

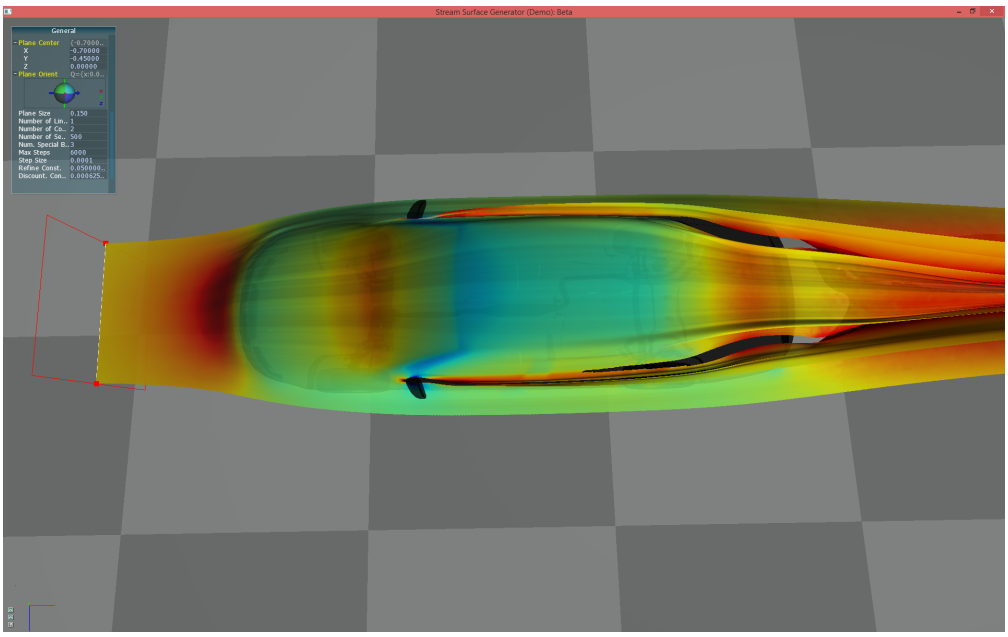


Figure 5.14.: **Streamsurface in car dataset**. ???

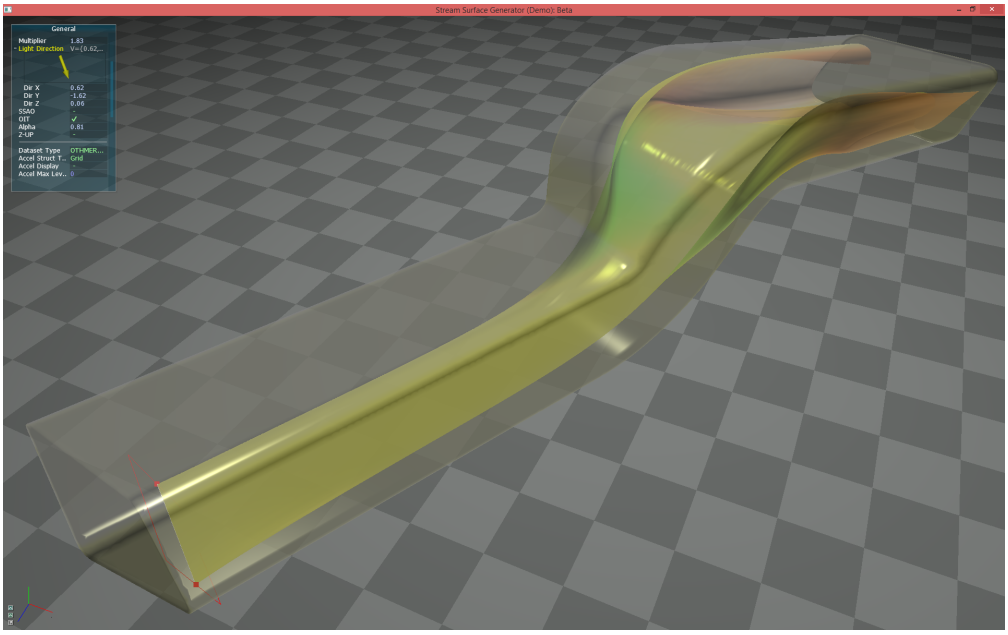


Figure 5.15.: **Streamsurface in wind tunnel dataset**. ???

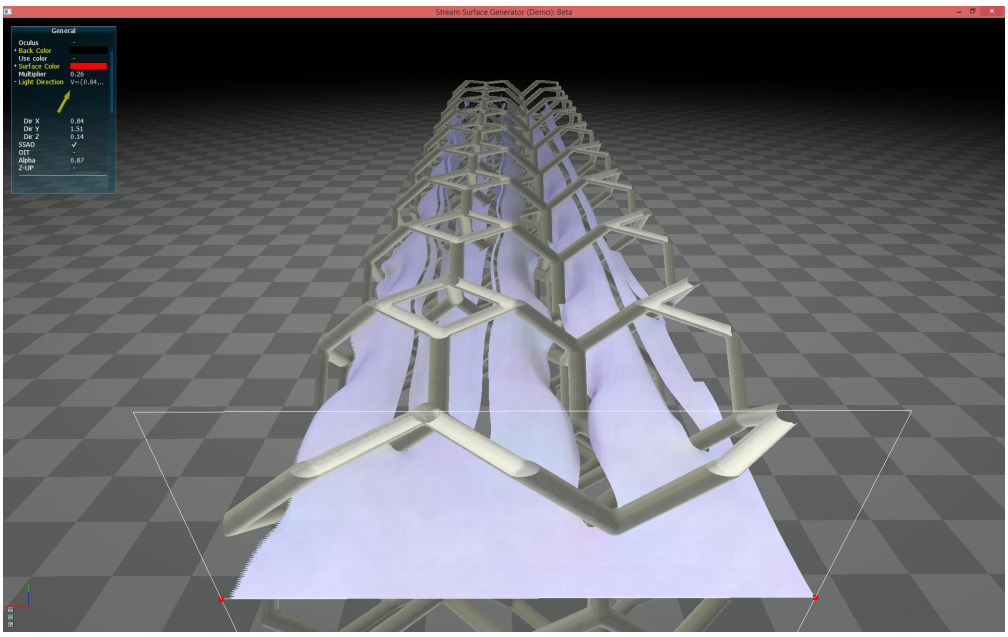


Figure 5.16.: **???**. ???

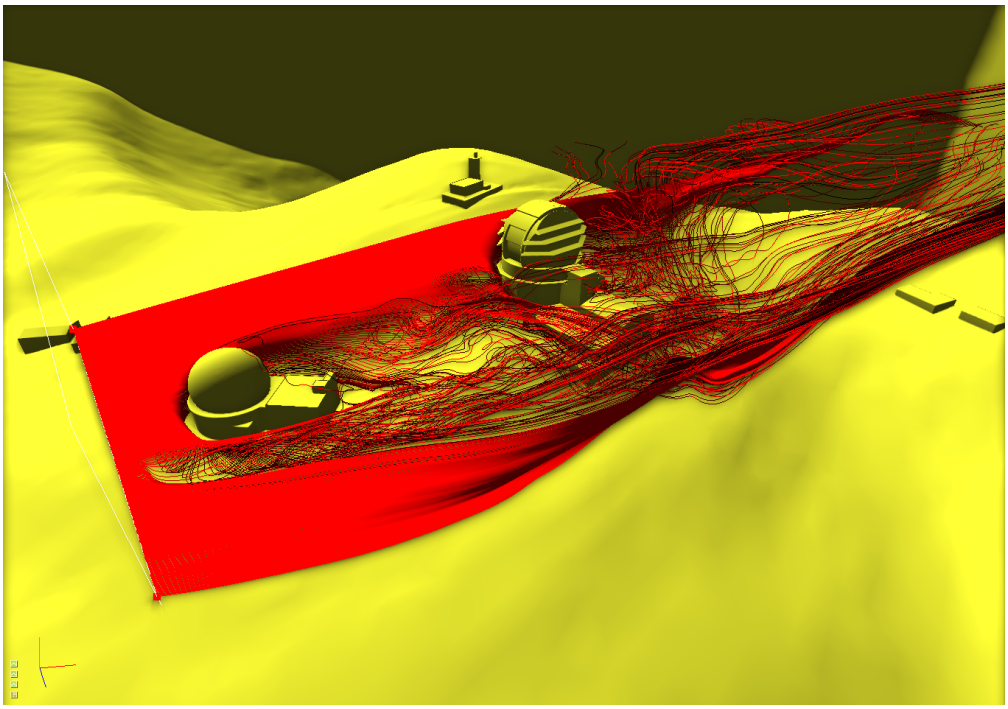


Figure 5.17.: **Streamlines in telescope dataset**. ???

6. Conclusion

Streamlines and streamsurface are the basic visualization tools used by engineers to get deep understanding into features of flow fields. In this master thesis, a heterogeneous streamline/streamsurface computation and rendering framework is proposed. The proposed algorithms can run on both CPU and GPU architectures. Therefore, all CPUs and GPUs of a system can be utilized for computations, and all GPUs for rendering. The concept of using acceleration structures for looking-up the velocity vector in a given point in the simulation mesh is applied instead of down-sampling simulation model to a regular grid which can reduce streamline/streamsurface computation. To improve streamline computation, parameters of acceleration structures, kernel execution model, and other techniques used in physically based rendering for performance improvement are evaluated and discussed.

6.1. Future Works

Chapter 5 presents evaluation of different acceleration structures. Although, the acceleration structure's performance are classified based on requirements, the hybrid acceleration structures for cell locating are not investigated, e.g. the evaluations show that a grid of BVH should have higher performance than normal BVH, while its size on the memory is less than normal grid acceleration structure. Additionally, it is shown that applying ray-packing concept to streamline computation can improve the long accurate streamline computation (large number of small integration steps). However, efficient streamline-packing require a decoupled streamline-sorter which re-packs the coherent streamlines that has divergence. Furthermore, OpenCL 2.0 has new features which can be utilized to get higher efficiency and performance e.g. shared virtual memory (SVM), and dynamic parallelism. The SVM can be used to share the dataset and the acceleration structures between devices which will reduce expensive data transfer cost, and make the framework memory efficient. Dynamic parallelism can be used to launch the refinement kernels on the device without taking the control back to the host. This is one of the poor refinement performance.

Appendices

A. OpenCL Programming Notes

In this appendix, a few points that are achieved during this master thesis about OpenCL programming is presented. It can be helpful for ones want to start with OpenCL.

- CodeXL is a great tool. Unfortunately, NVIDIA users cannot use it.
- In order to decrease the duration of kernel compilation, NVIdIA has implemented a caching scheme, where the already compiled kernels are stored on the disk. They will be reused in next compilations if the kernel does not change. The kernels are hashed from the source code to the index of the file in the cache. If the index was stored, the file will be reused, otherwise the compiler will compile the kernel. Unfortunately, the hashing procedure does not include the #*include* directives, which therefore the changes in the included files will not affect the compiled kernel. To solve this problem, the cache of the NVIDIA OpenCL compiler can be deleted. In window, the cache can be found in $\%APPDATA\%\backslash NVIDIA\backslash ComputeCache*$ and in linux, it can be found under */.nv/ComputeCache*. This problem can be solved also by preprocessing the kernel file, and stick the included files to the kernel file. *boost* :: *wave* is a good preprocessor which can be used in this context.

B. Syncing master/slave threads in BOOST by signaling

The goal of this appendix is to write a sample application which creates one main thread and ten slave threads. The slave threads must be run once after the main thread has started running. Handling the synchronization can be done via two different conditional variables. So, one is used for slave threads so they can wait until the main thread notify them and another conditional variable for the main thread which is signaled after each child finish its task, so the main thread can check if all the slave threads are done or not. The sample solution code for this problem is as follows:

```
#include <boost/atomic.hpp>
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <vector>

namespace /*static*/ {
  boost::atomic<int> data;

  boost::barrier* slave_thread_finished_barrier;
  boost::mutex slave_thread_finished_mutex;
  boost::condition_variable slave_thread_finished_cond;
  bool slave_thread_finished = false;

  struct Work {
    void signal_slave()
    {
      boost::lock_guard<boost::mutex> lock(data_ready_mutex);
      data_ready = true;
      cond.notify_all();
    }

    void slave_thread()
    {
      static boost::atomic_int _id_gen(0);
      id = _id_gen++;

      std::cout << "(" << id << "), slave_thread created\n";
```

```
      while (true) {

        boost::unique_lock<boost::mutex> lock(data_ready_mutex);
        cond.wait(lock, [&]{ return data_ready; });

        data_ready = false;

        data++;

        num_run++;

        slave_thread_finished_barrier->wait();

        // signaling the main thread that the slave threads are done.
        if (id == 0)
        {
          boost::lock_guard<boost::mutex> lock(slave_thread_finished_mutex);
          slave_thread_finished = true;
          slave_thread_finished_cond.notify_one();
        }

      }
    }

  private:
    int id = 0;
    bool data_ready = false;
    int num_run = 0;

    boost::mutex data_ready_mutex;
    boost::condition_variable cond;

  };

}

#include <boost/chrono.hpp>
#include <boost/chrono/chrono_io.hpp>

using hrc = boost::chrono::high_resolution_clock;

int main()
{
  boost::thread_group tg;

  size_t nThreads = 10;

  slave_thread_finished_barrier = new boost::barrier(nThreads);
```

```
  std::vector<Work> works(nThreads);

  for (size_t i = 0; i < nThreads; i++) {
    tg.create_thread(boost::bind(&Work::slave_thread, boost::ref(works[i])));
  }

  while (true) {
    auto start_time = hrc::now();

    for (auto& w : works)
      w.signal_slave();

    // Wait for slave threads to finish.
    boost::unique_lock<boost::mutex> lock(slave_thread_finished_mutex);
    slave_thread_finished_cond.wait(lock, [&]{ return slave_thread_finished; });
    slave_thread_finished = false;

    std::cout << "Elapsed␣Time␣=␣" << (hrc::now() − start_time) << std::endl;
  }

  tg.join_all();
}
```

Listing B.1: The basic idea behind the application presented in this master thesis which the main application thread handles the child threads synchronization.

List of Figures

List of Tables