LAZY WORK STEALING FOR CONTINUOUS HIERARCHY TRAVERSAL ON DEFORMABLE BODIES

Vinícius da Silva

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Março de 2013

LAZY WORK STEALING FOR CONTINUOUS HIERARCHY TRAVERSAL ON DEFORMABLE BODIES

Vinícius da Silva

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Claudio Esperança, Ph.D.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Luiz Henrique de Figueiredo, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2013

Silva, Vinícius da

Lazy Work Stealing for Continuous Hierarchy Traversal on Deformable Bodies/Vinícius da Silva. – Rio de Janeiro: UFRJ/COPPE, 2013.

X, 48 p.: il.; 29, 7cm.

Orientador: Claudio Esperança

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Bibliography: p. 45 – 48.

1. Continuous Collision Detection. 2. Deformable Bodies. 3. Hierarchy based methods. 4. GPGPU. I. Esperança, Claudio. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

LAZY WORK STEALING FOR CONTINUOUS HIERARCHY TRAVERSAL ON DEFORMABLE BODIES

Vinícius da Silva

Março/2013

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

Este estudo apresenta os resultados de pesquisa em balanceamento de carga dinâmico para Detecção de Colisão Contínua (CCD) usando Hierarquias de Volumes Envoltórios (BVHs) em Unidades de Processamento Gráfico (GPUs). A travessia de hierarquia é um problema desafiador para a computação em GPU, pois a carga de trabalho de percurso tem uma natureza muito dinâmica. Pesquisas atuais resultaram em métodos para dinamicamente balancear a carga conforme a travessia é executada. Infelizmente, a atual interface baseada em grade para computação em GPU não é totalmente adequada para este tipo de computação e o código de balanceamento de carga pode gerar sobrecarga excessiva. Este trabalho compara métodos de balanceamento de carga conhecidos, apontando prós e contras e apresenta um novo algoritmo para resolver alguns dos problemas mais notórios.

O algoritmo usa o novo conceito de roubo preguiçoso de trabalho, que tenta obter o máximo de paralelismo através de roubo ganancioso de trabalho e transferência preguiçosa. Além disso, o algoritmo é projetado para aumentar o uso de memória compartilhada por bloco e diminuir a penalidade de troca de contexto entre CPU e GPU.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

LAZY WORK STEALING FOR CONTINUOUS HIERARCHY TRAVERSAL ON DEFORMABLE BODIES

Vinícius da Silva

March/2013

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

This study presents the results of research in dynamic load balancing for Continuous Collision Detection (CCD) using Bounding Volumes Hierarchies (BVHs) on Graphics Processing Units (GPUs). Hierarchy traversal is a challenging problem for GPU computing, since the work load of traversal has a very dynamic nature. Current research resulted in methods to dynamically balance load as the traversal is evaluated. Unfortunatelly, current grid-based GPU computing interface is not well suited for this type of computing and load balancing code can generate excessive overhead. This work compares known load balancing methods, pointing pros and cons and presents a novel algorithm to address some of the most glaring problems.

The algorithm uses the new concept of lazy work stealing, which tries to get the most out of the parallel capabilities of GPUs by greedy work stealing and lazy work evaluation. Also, the algorithm is designed to augment shared memory usage per block and diminish CPU-GPU context exchange penalties.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Collision Detection has been a topic of great interest in computer graphics. The problem of detection of interfering objects appears in many applications such as robotics, games, and other simulations. The original Collision Detection approaches restricted the solution to the scope of Rigid Bodies discrete intersection. More recent investigation developed methods to address the problem in less restricted environments such as when objects are susceptible to deformable motion and when Continuous Collision Detection is considered. Also, work has been done to achieve better performance, exploiting some avenues of optimization techniques.

Several approaches have been proposed and they can be mainly divided in Bounding Volume Hierarchies, Distance Fields, Spatial Partitioning, Image-Space and Stochastic methods, as discussed in TESCHNER *et al.* (2004). Bounding Volume Hierarchy techniques organize the scene as a hierarchy of volumes that bound a group of objects, primitives or other volumes. The intersection against these volumes is easier to compute and this fact provide fast culling of parts of the scene. Distance Field is an implicit represention of objects which defines a field of distances for the entire proximity of the object. The object itself is defined at the zero level set, i. e. the places where the distance is equal to 0. Spatial Partitioning is the idea of subdividing the space in cells. Some approaches make this division based on the object itself, such as BSP trees, Octrees or Kd-trees, while others are object independent, such as Regular Grids. Image-Space techniques detect collisions using the projection of objects. Finaly, Stochastic methods are inexact and degrade the precision in favor of performance.

The present work documents the results of research in the area of load balancing for Continuous Collision Detection (CCD) using Bounding Volume Hierarchies (BVHs) on Graphics Processing Units (GPUs). Load balancing for GPUs is a challenge because current GPU programming interfaces are not well suited to problems with dynamically generated data or with uneven data distribution among processors. CCD using BVHs is such a problem, since the work load of BVH traversal is

very dynamic in nature.

The research about the aforementioned topic resulted in a novel approach to balance workload while traversing BVHs using GPUs. This document aims at describing this new idea and how it was conceived, providing the background concepts and the methodology to achieve the described results.

The next chapters are organized as follows. Chapter 2 comments all related work and introduces the concepts related to this dissertation. Chapter 3 focuses on the motivations and results in the proposal of a new method to address the presented problem. Chapter 4 discusses about the implementation details, shows the experiments and comparisons. Finally, Chapter 5 concludes this work, commenting about the limitations of the proposal and the directions for future work.

Chapter 2

Related Work

The present chapter is an introduction to the concepts that surround this dissertation, mainly Continuous Collision Detection (CCD) for Deformable Bodies (DBs) based on Bounding Volume Hierarchies (BVH) on Graphics Processing Units (GPUs). They are briefly explained in the next sections.

2.1 General Purpose Computing on GPU (GPGPU)

In recent years GPUs have evolved from configurable graphics processors to programmable graphics processors. This fact provided researchers and programmers with a cost-efficient many-core architecture for parallel computing (NICKOLLS and DALLY (2010)). Formerly, to use GPUs for generic computing, programmers should suit their problems to the programmable shaders pipeline. This activity requires representing generic data as graphic entities, such as textures, vertices and pixels, which is not user-friendly in some cases. The demand for generic programming in GPU has resulted in the development of GPGPU platforms and APIs, such as CUDA (NVIDIA (2012) , NICKOLLS *et al.* (2008)), OpenCL (GROUP (2012)) and Direct-Compute (NI (2009)). The nomenclature of concepts is different among APIs, so CUDA will be the focus from now on.

The CUDA architecture is based on Streaming Multiprocessors (SMs) . Each SM can run a predefined maximum number of threads that are grouped on two levels: warps and blocks. Warps are the low-level automatic grouping and are the SM scheduling unit. A warp executes one common instruction at a time, so full efficiency occurs when all threads in a warp agree on their execution path. If a data-dependent conditional divergent branch occurs, thread execution is serialized until convergence (NVIDIA (2012)).

Blocks are the high-level grouping and are defined by the programmer, who must

configure them depending on the problem and GPU resources available. Blocks are viewed by the GPU as a grid, so this is the way that a programmer must organize his problem and data for efficient solution using CUDA programs that are called *kernels*. Figure 2.1 shows the grid of thread blocks.

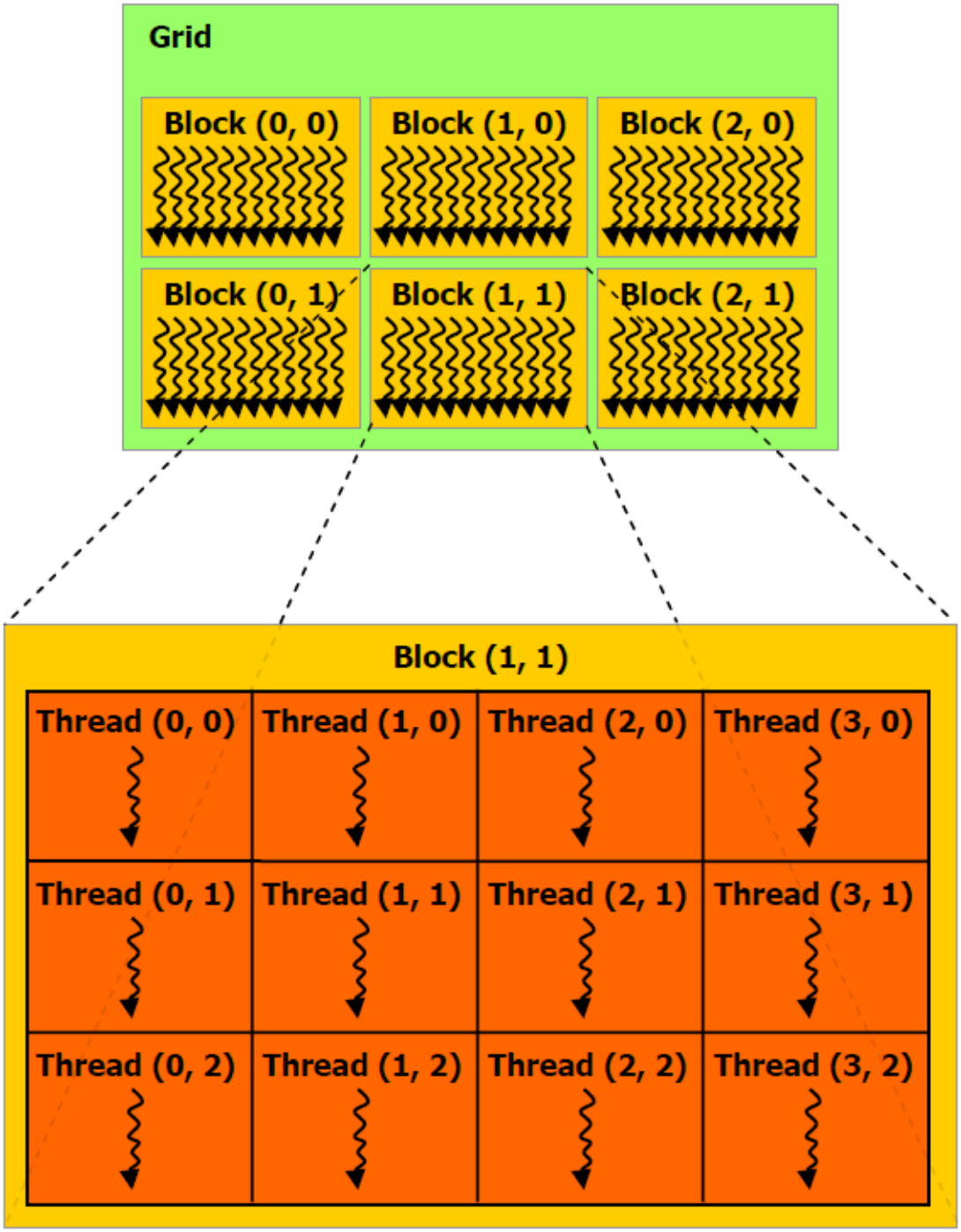

Figure 2.1: CUDA grid of thread blocks as shown in NVIDIA (2012)

CUDA devices have a memory hierarchy that reflects the logical organization of data inside the grid. It is divided in global, shared and local memory. The global memory is accessible by all threads in all blocks. It is more abundant, but has high latency. On the other hand, the shared memory is accessible inside a block. It is faster but scarcer. The local memory is accessible only inside a thread. Figure 2.2 shows the hierarchy.

2.2 Discrete and Continuous Collision Detection

Collision Detection (CD) is the research area of Computer Graphics that studies the problem of intersecting bodies in dynamic graphics simulations. For real-time environments, the simulations are supposed to be divided in frames f_i where the time interval between frames f_i and f_{i+1} is equal Δt_i (ERICSON (2004)). Several methods to approach the problem have been proposed and they can be classified as either Discrete Collision Detection (DCD) or Continuous Collision Detection (CCD).

DCD evaluates intersection by considering the positions of bodies at one single instant of each frame only, i.e., discarding what happens in any other instant in Δt_i. The algorithms are fast, but their correctness depends on Δt_i, i.e., collisions can be missed.

CCD evaluates intersection continuously in all of Δt_i. Thus, CCD methods correctness does not depend on Δt_i, but they are costly if compared with DCD methods. However, if GPGPU is taken into consideration, the performance cost of CCD methods can be used to mask memory latency, since CCD has heavier math than DCD. This aspect makes CCD cost-benefit on GPUs greater if a relation of culling efficiency and performance is considered.

2.3 BVHs

BVHs are a common used method to accelerate CD. The main idea behind this approach is to approximate geometry by a coarse representation called Bounding Volumes (BVs) , for which collision can be evaluated easily. These volumes spatially wrap a group of primitives or other BVs of a mesh. The BVs used naturally have a parent-child relationship and in consequence a tree can be used as a data structure to represent them. For shallow levels, the representation is coarser, while for deep levels the representation is more accurate. Using this approach the CD can benefit from fast geometry culling and space coherence, which is inherent to the problem. Figure 2.3 shows the concept.

BVHs can be classified by the type of BVs used. Oriented Bounding Boxes (OBBs) (GOTTSCHALK *et al.* (1996)) are reported as a good choice for use in

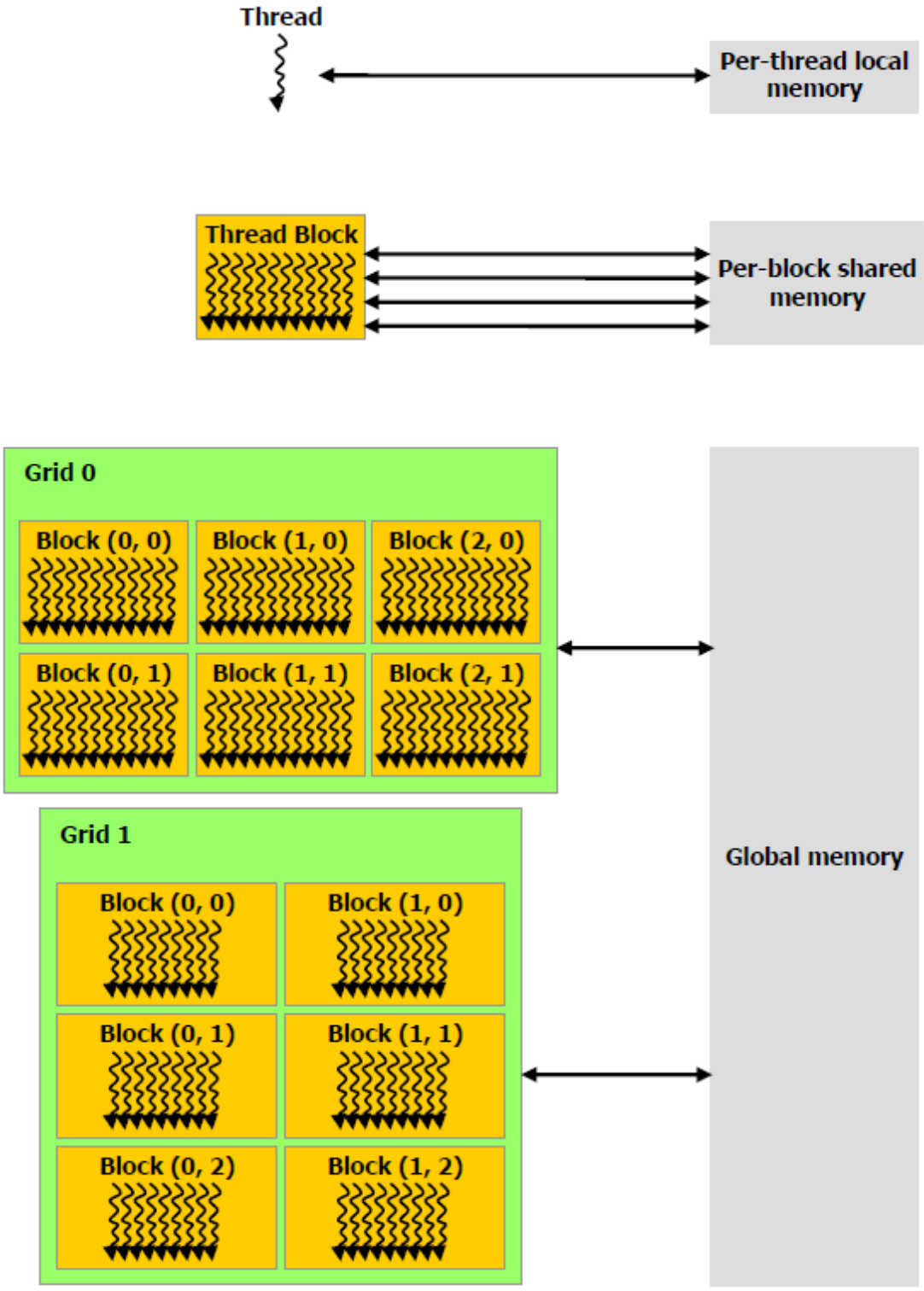

Figure 2.2: CUDA memory hierarchy as shown in NVIDIA (2012))

GPUs because they provide a better cost-benefit in respect with culling efficiency and computational overhead (LAUTERBACH *et al.* (2010)). The BVH data structure must be maintained during the whole simulation. This task is divided into BVH

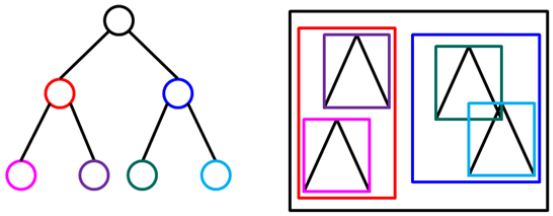

Figure 2.3: BVH concept (KIM *et al.* (2009))

creation, traversal for BV collision evaluation, primitive collision (elemental tests) and hierarchy update between frames. Several approaches can be used to accomplish each of these tasks.

2.3.1 Hierarchy Creation and Update

The methods for BVH construction consist of space division and are mainly top-down or bottom-up. Top-down approaches start with a BV enclosing all space. The BV is recursively divided until the number of enclosed primitives reaches a given predefined threshold. Bottom-up approaches start creating a BV for each primitive (or a predefined number of primitives) and each tree level is constructed by grouping BVs in the level immediately below. The differences between methods usually are the heuristics used to determine how to subdivide or to group bounding volumes and the number of allowed children for each parent. With respect of parallelism, top-down methods usually have less parallelism at the beginning of the construction, while bottom-up methods have less parallelism at the end of the construction.

The prototype associated with this dissertation creates OBBs by the usual covariance matrix eigenvalue method (GOTTSCHALK *et al.* (1996)). The mathematical proofs are beyond the scope of this text, but the process is briefly described here. For each node of the hierarchy the covariance matrix is generated using the primitive vertices if it is a leaf node or using the vertices of its children OBBs. The eigenvalues of the covariance matrix determine the OBB length, whereas the eigenvectors are used as the OBB axes. To solve the generated symmetric eigen system the approach described in PRESS *et al.* (2007) is used. First the Householder algorithm is applied to tridiagonalize the system matrix. Then, the QL algorithm with implicit shifts is used to finish the diagonalization. The eigenvalues are the remaining diagonal and the eigenvectors are formed by the accumulated transformation matrix of the QL algorithm.

The prototype discussed in this dissertation uses the approach described in LAUTERBACH *et al.* (2009) for hierarchy construction in GPU. It uses a grid subdivision of the space to generate a space-filling *Morton curve* and reduce the construction to a sorting problem. The prerequisites are previous knowledge of the

space boundaries of the entire geometry and the bounding volumes of all primitives (computed as described in the last paragraph). The space boundaries are used to compute how the space is quantized, i.e., the size of the grid node. The bounding volume *barycenter* of each primitive is used to compute its associated *Morton index code*, which indicates the node that contains the primitive. This code is formed by interleaved bits that indicate the side where the primitive is with respect to the division axis. Sorting primitives by their associated code also sorts them in the *Morton curve*. To achieve better parallelism, a radix sort by least significant bit algorithm is used to sort primitives. AJMERA *et al.* (2008) explains it in more detail. Figure 2.4 shows a Morton curve example, while figure Figure 2.5 represents the corresponding generated Morton codes.

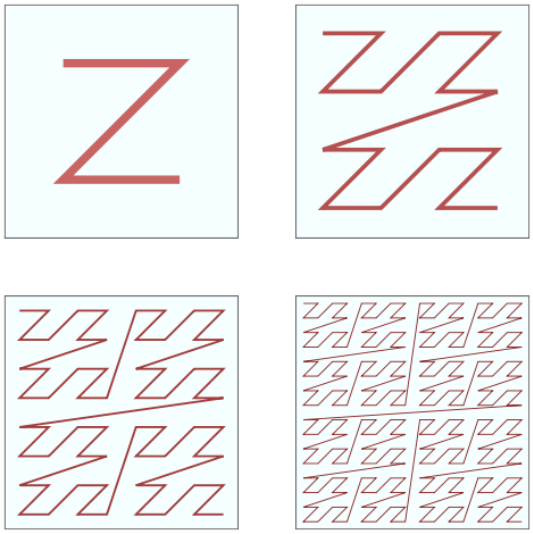

Figure 2.4: Morton curve

For each adjacent pair of primitives $i, i+1$ in the sorted sequence, their least common ancestor – the node farthest from the root whose subtree contains both – is determined by finding the most significant bit in which their *Morton codes* differ. If they differ in position h, a split among these primitives must be done at each hierarchy level from h to the max depth of the tree. This split list is recorded as $[(i,h),(i,h+1),...,(i,maxdepth)]$. The lists of each primitive pair are concatenated to form a split list sorted by the primitive index in the *Morton curve*. Then, this list is sorted by h (level). This final list has the information of how the primitives must be split at each level. The hierarchy can finally be constructed by linking child and parent nodes, removing side-effect singleton inner nodes, which can appear because split lists go until max depth, and updating the hierarchy so that each BV is big enough to enclose the primitives pointed by the split list (LAUTERBACH *et al.* (2009)).

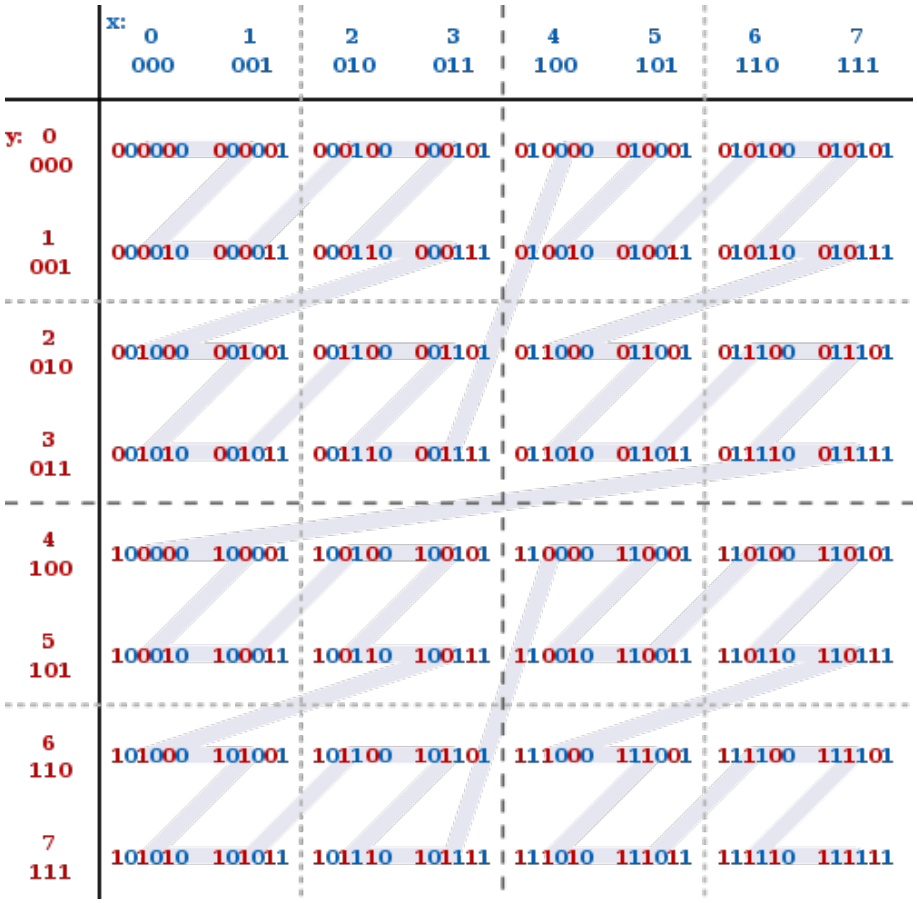

Figure 2.5: Morton code

There are several approaches to update the BVH. The prototype associated with this dissertation just refits each BV, recreating them with the new primitive or children OBBs vertex positions for the current frame.

2.3.2 Elemental Tests

Primitive pairs found to be potentially colliding must be evaluated with precise intersection elemental tests. In this section the elemental tests used among primitives are described in detail. The potentially primitive pairs are assumed to be already known and the process of finding them is not described in this section, but in section 2.3.3.

These tests depend on the type of mesh elements, and here we follow common practice by assuming triangular meshes, so the focus will be on continuous triangle×triangle intersection.

PROVOT (1997) described how to mathematically model the continuous triangle×triangle intersection problem and the discussion below follows in exactly same lines. There are two types of collision involved: a vertex intersecting a triangle (vertex-face collision) and an edge intersecting another edge (edge-edge collision).

A triangle×triangle intersection is reduced to 6 vertex-face tests (1 for each triangle vertex) and 9 edge-edge tests (each triangle edge against each edge in the other triangle). Let t_0 be the start of the time interval $[t_0, t_0 + \Delta t]$ and assume that the positions and velocities in t_0 are known. Assume also that the velocities are constant in the interval.

Vertex-face Collision

Let $P(t)$ be the vertex and $A(t), B(t), C(t)$ the vertices of the triangle. Let also $\vec{V}, \vec{V_A}, \vec{V_B}, \vec{V_C}$ be their respective constant velocities during the time interval. Thus, $A(t) = A(t_0) + t\vec{V_A}, B(t) = B(t_0) + t\vec{V_B}, C(t) = C(t_0) + t\vec{V_C}$. If there is collision, then

$$\begin{gathered} \exists t \in [t_0, t_0 + \Delta t] \text{ such that} \\ \exists u, v \in [0,1], u + v = 1, \vec{AP}(t) = u\vec{AB}(t) + v\vec{AC}(t) \end{gathered} \tag{2.1}$$

Equation 2.1 just shows that in the collision time t, P must be inside the triangle. But it is non-linear since u, v, t are unknown and there are factors that depend on two of them. To solve this, another condition is considered: that the triangle normal is orthogonal to the triangle. Thus

$$\vec{AP}(t) \cdot \vec{N}(t) = 0, \text{ where } \vec{N}(t) \text{ is the triangle normal.} \tag{2.2}$$

It is important to note that Equation 2.2 is not sufficient to verify the collision since it is true if A, B, C, P are coplanar. However it calculates t and therefore eliminates Equation 2.1 dependency on this variable and turns it linear. $\vec{N}(t)$ is a t^2 term and $\vec{AP}(t)$ is a t term, so Equation 2.2 is cubic. The approach to solve it is described in Section 2.3.2. Equation 2.2 can result in three values for t. The lowest positive value t' for which $P(t') \in ABC(t')$ is chosen as the final result.

Edge-edge Collision

The ideas in Section 2.3.2 can be used in the edge-edge collision case with minor changes. Let $AB(t)$ be one edge and $CD(t)$ be the other one. The collision occurs if and only if

$$\begin{gathered} \exists t \in [t_0, t_0 + \Delta t] \text{ such that} \\ \exists u, v \in [0,1], u\vec{AB}(t) = v\vec{CD}(t) \end{gathered} \tag{2.3}$$

Once again, this is a nonlinear system. The relation used to calculate t is that A, B, C, D must be coplanar, like before.

$$(\vec{AB}(t) \times \vec{CD}(t)) \cdot \vec{AC}(t) = 0 \tag{2.4}$$

This also is a cubic equation, which can lead to 3 values for t. The lowest positive value that makes it possible for Equation 2.3 to be solved is chosen as the final result.

Cubic Equations Solver

The prototype associated with this dissertation uses the combination of Newton-Raphson and bisection methods as described in PRESS *et al.* (2007). The bisection method gets an interval $[a, b]$, where the root is known to be, i.e., $f(a)$ and $f(b)$ have opposite signals. The algorithm iterates by dividing the interval at the midpoint, reducing interval to $[a', b']$ as a result, and evaluating $f(a')$ and $f(b')$. This is done until the error in the root value is acceptable. The bisection method is assured to find the root, but has slower convergence than other methods. On the other hand, the Newton-Raphson uses the derivative of the function to refine a root guess. If f is the function and a is the root guess, the Newton-Raphson method iterates by evaluating the zero crossing of the tangent line of f passing through $f(a)$. The new root guess will be the abscissa of this zero crossing. Figure 2.6 shows the method. It converges fast, but has some special cases that can lead to divergence or cycling.

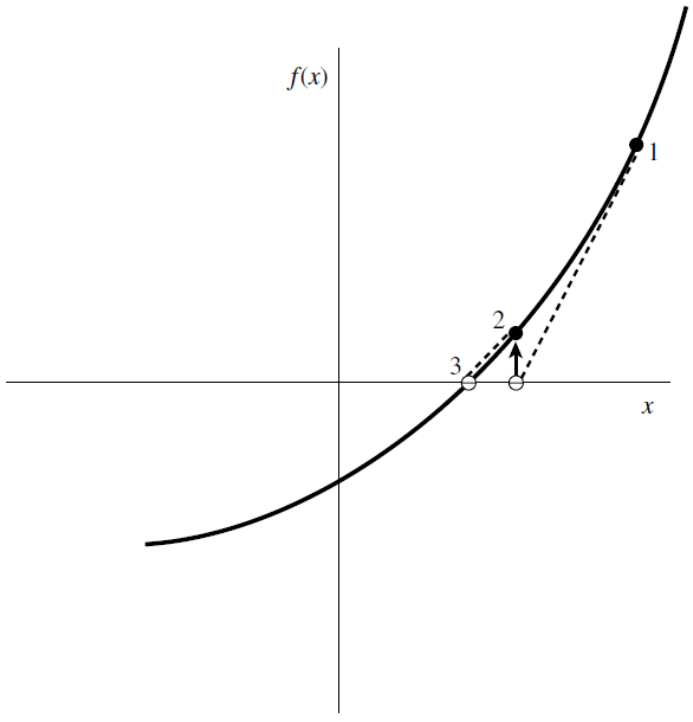

Figure 2.6: Newton-Raphson method, as in PRESS *et al.* (2007)

The approach of PRESS *et al.* (2007) suggests using Newton-Raphson while it is converging fast enough and bisection otherwise. This allows fast and safe convergence.

Culling Methods

There are several culling methods for primitive pair collision. They aim to eliminate false positives and redundant computations. In the prototype of this dissertation the Non-penetration Filter algorithm (TANG *et al.* (2010a)) is used. It eliminates the need for solving the cubic equation of a vertex-face pair (p, t) when point p remains on the same side of triangle t during all time interval. Similarly, an edge-edge pair can be culled if they cannot be coplanar during the time interval. These relations were originally presented as theorems and are reproduced here for convenience, with their consequences explained later. The mathematical proofs of the theorems are beyond the scope of this work, but can be viewed in the aforementioned paper.

For the vertex-face case, consider a vertex P_t and a triangle T_t. The projected distance between them along the normal n_t indicates if they can be coplanar (Figure 2.7).

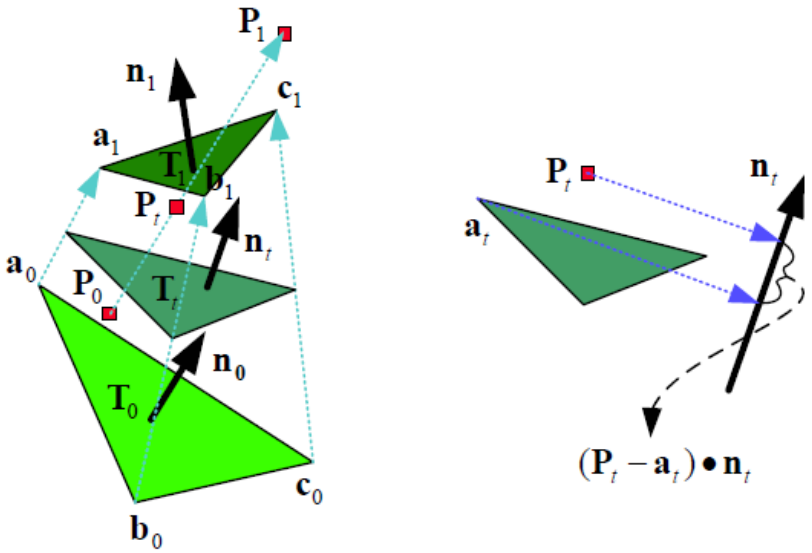

Figure 2.7: Deforming triangle, vertex and projected distance, as shown in TANG *et al.* (2010a)

Non-coplanar Theorem for VF tests: *For a triangle T_t and a vertex P_t defined by the start and end positions during the interval [0, 1], these positions are linearly interpolated in the interval with respect to the time variable, t. If the following four scalar values: $A, B, \frac{2*C+F}{3}$ and $\frac{2*D+E}{3}$ have the same sign, T_t and P_t will not be coplanar during the interval:*

$$A = (P_0 - a_0) \cdot n_0,\ B = (P_1 - a_1) \cdot n_1,\ C = (P_0 - a_0) \cdot \hat{n},$$
$$D = (P_1 - a_1) \cdot \hat{n},\ E = (P_0 - a_0) \cdot n_1,\ F = (P_1 - a_1) \cdot n_0$$

And:

$$n_0 = (b_0 - a_0) \times (c_0 - a_0), n_1 = (b_1 - a_1) \times (c_1 - a_1)$$

$\hat{n} = \frac{(n_0+n_1-(\vec{v_b}-\vec{v_a})\times(\vec{v_c}-\vec{v_a})}{2}$
$\vec{v_a} = a_1 - a_0, \vec{v_b} = b_1 - b_0, \vec{v_c} = c_1 - c_0$

TANG *et al.* (2010a) show that the projected distance of P_t and T_t is:

$$(P_t - a_t) \cdot n_t = A * B_0^3(t) + \frac{2 * C + F}{3} * B_1^3(t) + \frac{2 * D + E}{3} * B_2^3(t) + B * B_3^3(t), \quad (2.5)$$

where $B_i^3(t)$ is the i^{th} basis function of the Bernstein polynomials of degree 3.

Equation 2.5 is a Bézier curve and the values $A, B, \frac{2*C+F}{3}$ and $\frac{2*D+E}{3}$ are its control vertices. Since they are scalars, they can be thought of as delimiting an interval in the Real axis. The curve can be constructed varying t. The convex hull property of the control points states that any generated curve point is bounded by them. Thus, if $A, B, \frac{2*C+F}{3}$ and $\frac{2*D+E}{3}$ have the same sign, the projected distance cannot be equal to 0 in the time interval and no collision occurs.

The same concepts can be extended to the edge-edge collision. Consider two edges E^1 and E^2 (defined by u_0, v_0 and k_0, l_0 at $t = 0, u_1, v_1$ and k_1, l_1 at $t = 1$). Figure 2.8 shows the concepts.

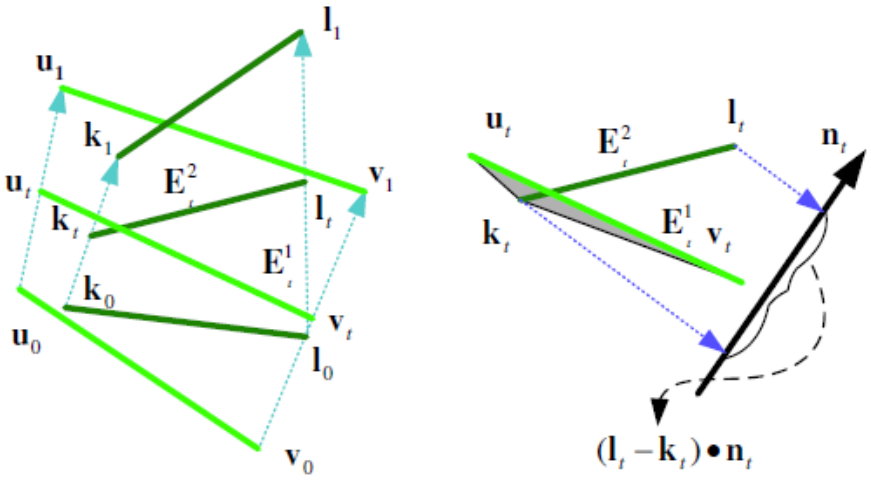

Figure 2.8: Deforming edges and projected distance, as shown in TANG *et al.* (2010a)

Non-coplanar Theorem for EE tests: *For two deforming edges E^1 and E^2 defined by the start and end positions during the interval [0, 1], these positions are linearly interpolated in the interval with respect to the time variable, t. If the following four scalar values: $A', B', \frac{2*C'+F'}{3}$ and $\frac{2*D'+E'}{3}$ have the same sign, E^1 and E^2 will not be coplanar during the interval:*

$A' = (l_0 - k_0) \cdot n'_0,\ B' = (l_1 - k_1) \cdot n'_1,\ C' = (l_0 - k_0) \cdot \hat{n'},$
$D' = (l_1 - k_1) \cdot \hat{n'},\ E' = (l_0 - k_0) \cdot n'_1,\ F' = (l_1 - k_1) \cdot n'_0$
and
$n'_0 = (u_0 - k_0) \times (v_0 - k_0), n'_1 = (u_1 - k_1) \times (v_1 - k_1)$
$\hat{n'} = \frac{(n'_0 + n'_1 - (\vec{v_u} - \vec{v_k}) \times (\vec{v_v} - \vec{v_k})}{2}$
$\vec{v_k} = k_1 - k_0, \vec{v_u} = u_1 - u_0, \vec{v_v} = v_1 - v_0$

As in Equation 2.5, the projected distance can be represented as a Bézier curve using the new control points and the coplanarity tests can be done in the same way.

2.3.3 BVH Traversal

The tree traversal is the operation that actually evaluates collision and generates a potentially colliding primitive pair list. This list is refined later by more precise elemental tests as described in Section 2.3.2. The traversal can evaluate inter-collision, intra-collision or both. Inter-collision occurs when different trees or subtrees intersect. On the other hand, intra-collision occurs when the intersection happens in the same tree or subtree. Algorithm 2.1 shows the traversal.

It is important to note that intra-collision is much more expensive to compute than inter-collision, mainly because adjacent primitives can lead to a huge number of false-positives. This fact is a consequence of the use of conservative representations such as BVs. Several culling heuristics have been proposed to address this problem (HEO *et al.* (2010) , TANG *et al.* (2008)).

The prototype of this dissertation uses the concept of Orphan Sets defined in TANG *et al.* (2008) to ignore adjacent triangle intersections in traversal. The method relies in a preprocessing step on the mesh geometry to find all elemental tests generated by adjacent triangle pairs that cannot be generated by non-adjacent triangle pairs. Two auxiliary concepts are used to generate the Orphan Sets: Orphan Incident Set (OIS) and Orphan Adjacent Set (OAS) . OIS is the set of incident triangles of a mesh feature, while OAS is the set of adjacent triangles of a mesh feature. These concepts are illustrated in Figure 2.9.

There are two types of orphan features: boundary and interior. Each vertex and edge on the boundary of a mesh will be orphan as shown in Figure 2.9. Interior orphans can arise from some special cases, in which every triangle is adjacent to every other triangle or in triangle fans. Figure 2.10 shows such cases.

Algorithms 2.2 and 2.3 are used to create the Orphan Set for vertices and edges respectively. The generated orphan tests can be refined using other techniques, such as the Continuous Normal Cone (CNC) (TANG *et al.* (2008)) or feature bounding volumes or they can be concatenated in the elemental tests list generated after each traversal. It is important to remember that the triangle bounding volumes cannot be

Algorithm 2.1 BVH inter and intra collision traversal. In both cases it is assumed that the root node is bounding the entire scene.

```
1: function traverseIntra(bvh)
2:     node0 ← bvh.root
3:     node1 ← bvh.root
4:     traverseTree(node0 , node1)
5: end function

6: function traverseInter(bvh)
7:     node0 ← bvh.root.leftChild
8:     node1 ← bvh.root.rightChild
9:     traverseTree(node0 , node1)
10: end function

11: function traverseTree(node0, node1)
12:     if node0 = node1 then                                   ▷ Intra-collision
13:         left ← node0.leftChild
14:         right ← node0.rightChild
15:         traverseTree(left, right)
16:         if NOT left.isLeaf then
17:             traverseTree(left, left)
18:         if NOT right.isLeaf then
19:             traverseTree(right, right)
20:     else                                                    ▷ Inter-collision
21:         BV0 ← node0.bv
22:         BV1 ← node1.bv
23:         if collide(node0 , node1) then
24:             if node0.isLeaf AND node1.isLeaf then
25:                 primitivePair ← (node0.primitive, node1.primitive)
26:                 primitivePairList.addPair(primitivePair)
27:             else
28:                 if node0.isLeaf OR BV1.volume > BV0.volume then  ▷ Node1
    is greater, check its children
29:                     traverseTree(node0, node1.leftChild)
30:                     traverseTree(node0, node1.rightChild)
31:                 else                          ▷ Node0 is greater, check its children
32:                     traverseTree(node1, node0.leftChild)
33:                     traverseTree(node1, node0.rightChild)
34: end function
```

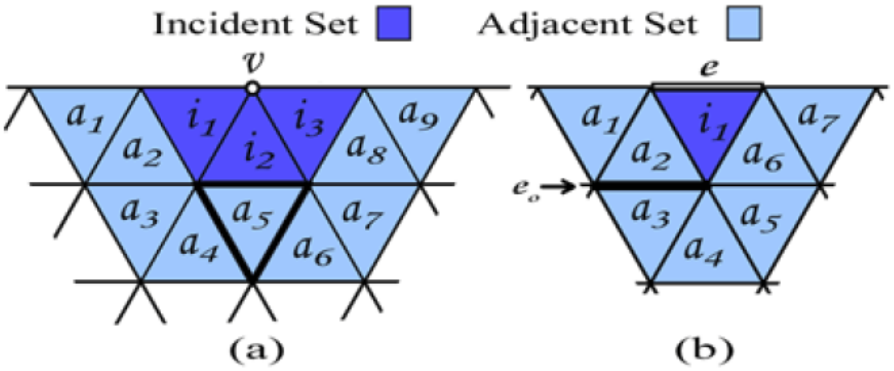

Figure 2.9: Boundary orphan features. (a) shows orphan vertex v, which makes an orphan pair with face a_5. (b) shows edge e, which makes an orphan pair with e_0 and other features. Picture reproduced from TANG *et al.* (2008).

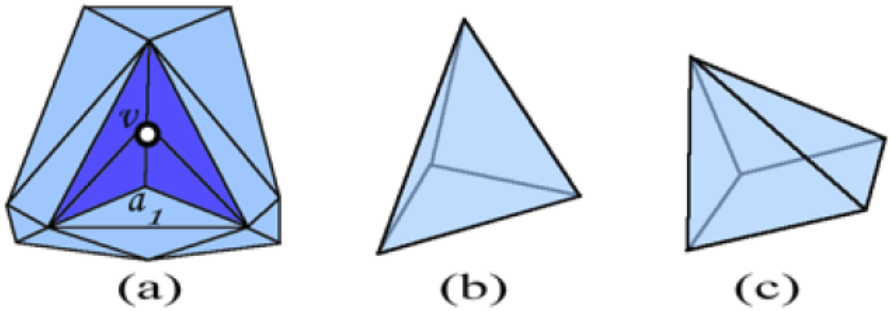

Figure 2.10: Internal orphan features. (a) shows vertex v, which is orphan because it is the shared vertex of the triangle fan, (b) a tetrahedron, where each vertex and opposite face form an orphan pair. The same happens with the 4-sided pyramid (c). As shown in TANG *et al.* (2008).

used to refine the tests since they would always point to a collision among adjacent triangle pairs.

The prototype of this dissertation also implements the concept of front-based decomposition (TANG *et al.* (2010b)). This concept allows the usage of time coherence in BVH traversal by saving the front of the Bounding Volume Test Tree (BVTT) related with the traversal. The BVTT is the tree that represents the collision tests performed in the traversal and the front consists of the BVTT nodes where the traversal ends. The front can be used as the starting point for the traversal in the next frame, saving computational time and providing more parallelism. The front can be acquired while in traversal by saving the pairs of BVH nodes in which the traversal ends. These pairs are those whose collision is evaluated as false or leaf pairs. The concepts are shown in Figure 2.11. The blue nodes are the original BVH. The yellow nodes are the test tree for an arbitrary intra-collision traversal. The front is marked in gray and is formed by all colliding or non-colliding BVH leaf pairs and any other pair that is non-colliding.

Algorithm 2.2 FindVFOrphan: Find all orphan pairs built on vertex v as showed in (OrphanSets)

```
1: Create OIS for v
2: Create OAS for v
3: for all Triangle t_a ∈ OAS do
4:     adjacent = TRUE
5:     for all Triangle t_i ∈ OIS do
6:         if t_a not adjacent t_i then
7:             adjacent = FALSE
8:             break
9:     if adjacent == TRUE then
10:        Add orphan pair {v, t_a}
```

Algorithm 2.3 FindEEOrphan: Find all orphan pairs built on edge e as showed in (OrphanSets)

```
1: Create OIS for e
2: Create OAS for e
3: Create ES, the set of all edges in OAS
4: for all Edge e_a ∈ ES do
5:     if e_a is incident to e then
6:         continue
7:     Create OIS_a for e_a
8:     adjacent = TRUE
9:     for all Triangle pair {t_i, t_a}, t_i ∈ OIS & t_a ∈ OIS_a do
10:        if t_a not adjacent t_i then
11:            adjacent = FALSE
12:            break
13:    if adjacent == TRUE then
14:        Add orphan pair {e, e_a}
```

The front saved from previous frame f_p can be used as the starting point for the traversal in the current frame. This operation is called front tracking and is shown in Figure 2.12. To execute it, the collision must be evaluated for each front node (x, y) in f_p. If x and y are both BVH leaf nodes, they must be added in the current frame front f_c, since leafs are in the front by definition, and their triangles must be added to the potentially colliding triangle pair list if their BVs collide. If x or y is a BVH internal node, then if x and y remain non-colliding, they are just added to f_c. Otherwise, the pair (x, y) is added to the traversal work list so the BVTT traversal is resumed from this node.

The prototype associated with this dissertation uses an heuristic to evaluate when the BVTT front is considered outdated and must be rebuilt. It consists of comparing the number of nodes in the front $n(f_c)$ at the current frame f_c and the number of nodes in the front $n(f_{c-p})$ at a reference previous frame f_{c-p}. If $\frac{f_c}{f_{c-p}} < rt$, where rt is a rebuild threshold, the front does not have changed too much at the last frames and is considered outdated, i.e., it is too deep for the current scene state and in consequence is generating more computations than the desirable. The best values for p and rt are model dependent. Rebuilding the front means traversing the hierarchy againg from the root.

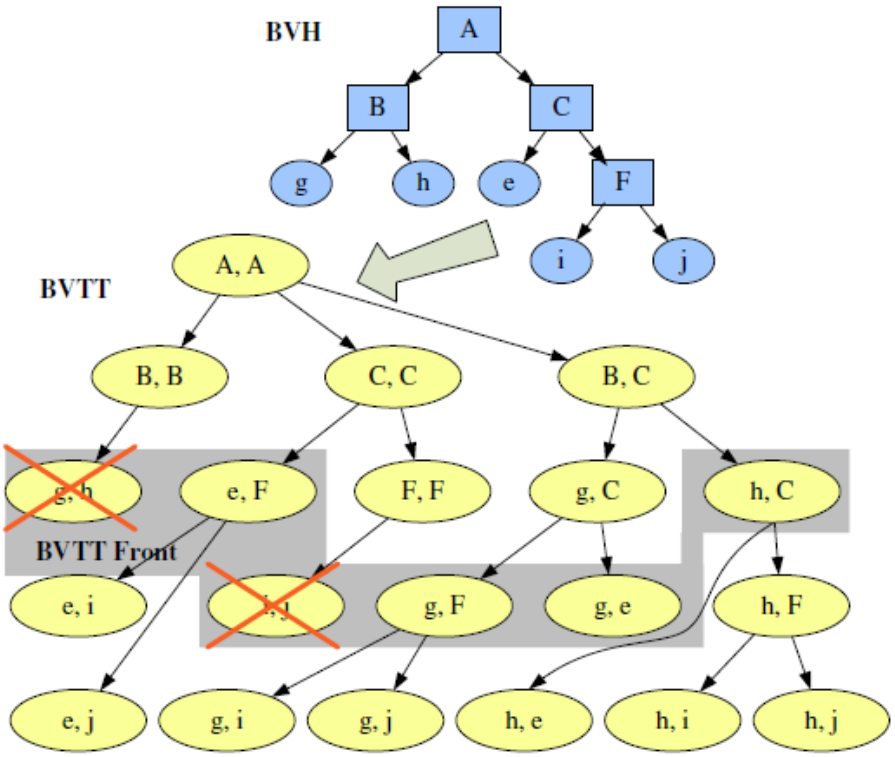

Figure 2.11: BVH and associated BVTT for an intra-collision traversal. The marked nodes in the BVTT are the front, i.e., the nodes where the collision traversal ended. As shown in (TANG *et al.* (2010b)).

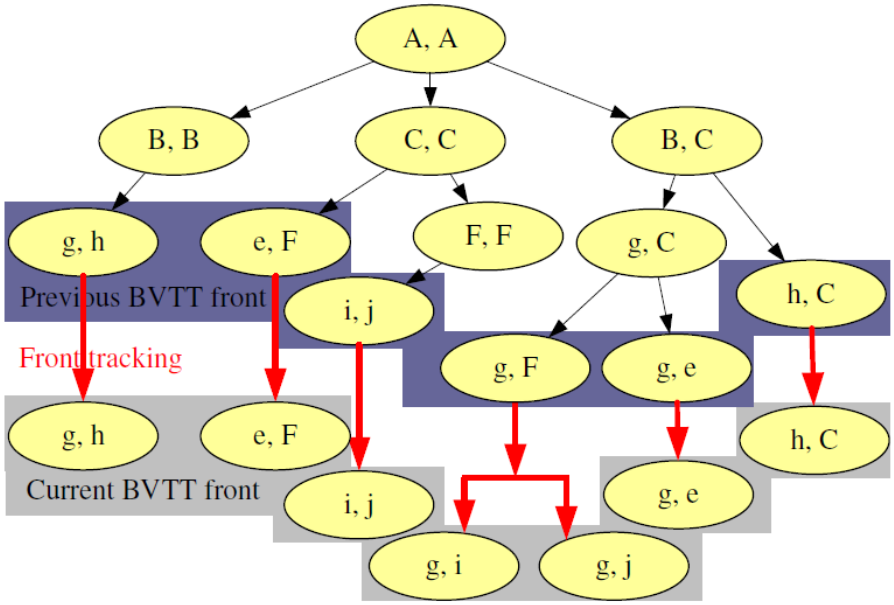

Figure 2.12: Front tracking representation. The front saved from previous frame is used as the starting point for traversal in the current frame. This traversal generates the front used in the next frame. This process continues until the front is considered outdated and rebuilt. As shown in (TANG *et al.* (2010b)).

Chapter 3

Hierarchy Traversal Load Balancing

The main difficulty that arises when designing CCD BVH algorithms for GPUs is the fact that workload can be generated on-the-fly during hierarchy traversal. Usually, problems are better suited for running on GPUs when the workload is completely known a priori and thus can be evenly divided among GPU cores before kernel calls. However, researchers have studied this load balancing problem and have proposed approaches for achieving better performance and flexibility. In particular, four different approaches for load balancing on GPUs are worthy of mention: Static Task List, Blocking Dynamic Task Queue, Lock-free Dynamic Task Queue and Task Stealing. A discussion about these approaches can be found in (CEDERMAN and TSIGAS (2009)).

This chapter focuses on the concepts of Static Task List and Task Stealing and how they can be used for hierarchy traversal. A novel load balancing algorithm for hierarchy traversal is presented, based on the *task stealing* load balancing approach discussed in CEDERMAN and TSIGAS (2009). The algorithm is also compared with the load balancing scheme used in gProximity (LAUTERBACH *et al.* (2010)).

3.1 Static Task List

This method is the simplest load balancing scheme mentioned in CEDERMAN and TSIGAS (2009). It does the balancing before issuing all work and is inherently inflexible because of this. In terms of dynamically generated workload, this scheme can lead to extremely unbalanced cases when all cores finish their issued work quickly except a core which has the most time-consuming task. From the point of view of hierarchy traversal, this case arises when one core needs to perform a deep hierarchy traversal while all others finish their traversal fast. In this case, we have almost

all cores in an idle state, and thus wasting useful resources. Specifically, let (a_i, b_i) denote a hierarchy node pair which is to be processed in core i, where $0 \leq i < N$, N being the number of available cores. Also, suppose function $C(x, y)$ evaluates collision between the nodes x and y. Clearly, if $C(a_0, b_0) = \text{true}$ and $C(a_i, b_i) = \text{false}$ $\forall i \neq 0$, then core 0 will have all the work thereafter and the other cores will be idle.

LAUTERBACH *et al.* (2010) proposed an algorithm to address the inflexibility of the Static Task List method. The idea is to redistribute nodes if the number of unused cores is more than a predefined threshold. Each core has its own stack where nodes can be pushed in or popped out. Using CUDA terminology, in this algorithm a core is defined as a CUDA block and the traversal and load balancing kernels are used to fulfill the related tasks. The threshold of unused cores is passed as a parameter for the traversal kernel. In this section, the terms "block" and "core" are used interchangeably.

The algorithm proceeds on two levels: CPU and GPU. The CPU code stays in a loop, calling two CUDA kernels in sequence: one for hierarchy traversal and another for load balancing. This continues until all the traversal is done. A scheme of how kernels correlate, as presented in LAUTERBACH *et al.* (2010), is depicted in Figure 3.1. The algorithms are presented in Algorithms 3.1 to 3.4. All kernel parameters are assumed to be in GPU global memory. Also, the function $assignShared(sharedData, value)$ is suposed to use one thread to assign $value$ to a block shared variable $sharedData$. Finally, all SYNC calls are supposed to do the synchronization of threads inside a block.

In the traversal kernel, a block stays in a loop until stalled by the condition of excessive number of idle blocks or if its stack is empty (no more nodes) or full (no more space to generate new nodes). The number of unused cores is maintained by a global counter. The loop consists first of verifying the global counter and stalling if the condition of excessive unused blocks is met. Otherwise the block pops traversal nodes from its stack to feed threads, processes them, possibly generating new nodes on-the-fly, and pushes the new nodes back to its stack. Afterwards, the block checks the condition of its stack being empty or full and stalls if it is the case. A block stall means incrementing the global counter of unused blocks, pushing all remaining nodes on local stack to a global memory list and returning all threads in block. The global memory list is segmented and each block has its own space to save its stack. It is important to note that blocks that still have useful nodes are forced to stall if the excessive unused cores condition is met. The algorithm is described in Algorithm 3.2.

The second kernel balances the load, receiving as input the list of stack nodes saved on the segmented global memory and the list of counters per stack from previous traversal. The output is a new list which will have the nodes rearranged evenly

and a list of counters that reflects the changes. To achieve this, the kernel first copies the list of stack counters to shared memory. Then it does a parallel reduction operation (NGUYEN (2007)) which computes the total of nodes. Afterwards, the number of cores per stack is calculated by dividing the total of nodes by the number of cores. Finally, for each stack in the segment input list, node positions are rearranged so all stacks have the same number of nodes. The algorithm declarations are described in Algorithm 3.3, while the kernel is described in Algorithm 3.4.

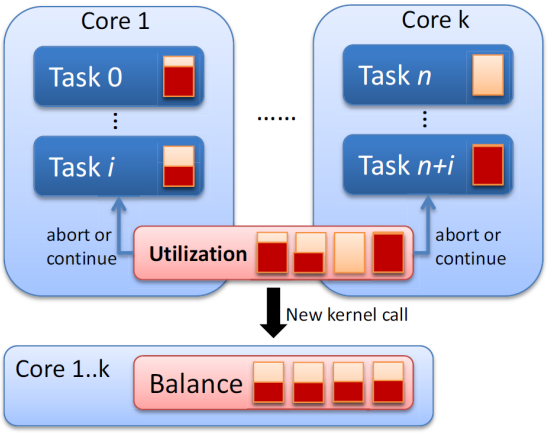

Figure 3.1: LAUTERBACH *et al.* (2010) load balancing for hierarchy traversal scheme.

Algorithm 3.1 gProximity traversal and load balancing - CPU level

```
1: while gTotalNodes > 0 do
2:     TRAVERSETREE(gStacks , gTriPairs , gIdleCores)
3:     BALANCEWORKLIST(gStacks , gOutStacks , gTotalNodes)
4:     SWAP(gStacks , gOutStacks)
```

3.2 Task Stealing

According to CEDERMAN and TSIGAS (2009), task stealing is a popular load balancing scheme where each processing unit is given a set of tasks and when it has completed them it tries to steal a task from another processing unit which has not yet completed its assigned tasks. If a unit creates a new task it is added to its own local set of tasks.

The scheme studied in the aforementioned paper is an adaption for GPUs of ARORA *et al.* (1998) task stealing method using non-blocking double ended queues (deques). It was designed in the scope of multiprocessor systems so we will define *process* as an instance of a computer program which can run one thread at a time. Also, it assumes that processes can run in parallel or concurrently. Each process has

Algorithm 3.2 gProximity traversal kernel - GPU level

```
1: function traverseTree(gStacks , gTriPairs , gIdleCores)
2:     Parameters (global memory):
3:         gStacks                                          ▷ BVTT stacks
4:         gTriPairs                          ▷ Triangle pairs for elemental tests
5:         gIdleCores                                       ▷ Number of idle cores
6:     Block shared memory variables:
7:         sWorkCount                                       ▷ Deque work element count
8:         sAborted                          ▷ gIdleCores as visible for this block
9:     Thread local memory variables:
10:        block                                            ▷ Block index in the grid
11:        thread                                           ▷ Thread index in the grid
12:        nThreads                                         ▷ Number of threads per block
13:        nActive          ▷ Number of active threads for the current iteration
14:        workItem                                         ▷ BVTT node
15:    assignShared(sWorkCount , gStacks[block].count)
16:    sync                          ▷ Synchronizes with threads in the same block.
17:    if sWorkCount = 0 then
18:        callAbort(gIdleCores)                            ▷ Increment gIdleCores once
19:        return
20:    sync
21:    while sWorkCount > 0 do
22:        nActive ← min(nThreads, sWorkCount)
23:        workItem ← INVALID
24:        if thread < sWorkCount then
25:            workItem ← gStacks[block].deque[sWorkCount − nActive + thread]
26:        sync
27:        assignShared(sWorkCount , sWorkCount − nActive)
28:        sync
29:        if workItem ≠ INVALID then
30:            checkCollision(workItem.x , workItem.y)      ▷ Can generate nodes
31:        sync
32:        if (sWorkCount >= THRESHOLD) OR (sWorkCount = 0) then
33:            callAbort(gIdleCores)                        ▷ Abort if stack is full or empty
34:            break
35:        assignShared(sAborted , gIdleCores)
36:        sync
37:        if sAborted > CORES_FOR_ABORT then
38:            callAbort(gIdleCores)                        ▷ Abort if have too many idle cores
39:            break
40:    assignShared(gStacks[block].count , sWorkCount)
41: end function
```

Algorithm 3.3 gProximity balancing kernel declarations - GPU level.

```
1:  Parameters (global memory):
2:      gStackIn                                          ▷ Input stack
3:      gStackOut                                         ▷ Output stack
4:      gTotalNodes                          ▷ Output the total number of nodes
5:  Block shared memory variables:
6:      sNThreads                            ▷ Number of threads per block
7:      sSum[nThreads]                                    ▷ Prefix sum array
8:      sSizes[nThreads]                                  ▷ Stack sizes
9:  Thread local memory variables:
10:     block                                ▷ Block index in the grid
11:     thread                               ▷ Thread index in the grid
12:     nodesLeft                  ▷ Number of total nodes left to be copied
13:     nodesPerStack              ▷ Nodes per stack (nodesLeft/sNThreads)
14:     iOffset                              ▷ Input stack offset
15:     oOffset                              ▷ Output stack offset
16:     inStack                              ▷ Input stack reference
17:     nodesLocal   ▷ Number of left nodes to be copied in the current inputStack
18:     nodesToWrite ▷ Number of nodes to be copied to the current output stack
```

its own local thread deque and can acquire threads by popping from it. A process can generate work on-the-fly by pushing threads on its local deque. Each process stays in a loop, popping a thread from its local deque and doing computations by dispatching the assigned thread. If the local deque is empty, a process yields the processor to avoid starving others and steals work from the deque of another process. To minimize parallelization penalties, steals are done by popping from *top* pointers while usual thread acquisition is done on local deque *bottom* pointers. In addition, work is generated by pushing local bottom pointer only. The algorithm finishes when there is no more work to do. Algorithms 3.5 and 3.6 explain this process in more detail.

Since this algorithm is lock-free there are some details on the deque manipulation inner functions, namely *pushBottom()*, *popBottom()* and *popTop()*, that must be noted. First, all operations on deque *top* pointers can result in race conditions and are resolved by atomic operations. On the other hand, operations on bottom (*bot*) pointers are fully parallel unless the deque pointers must be reset in consequence of emptiness, since just the block that owns the deque is allowed to change *bot*. The algorithm uses Compare And Swap (CAS) instructions to handle the race conditions (HERLIHY (1991)).

The compare-and-swap instruction cas operates as follows. It takes three operands: a register addr that holds an address and two other registers, old and new, holding arbitrary values. The instruction cas(addr, old, new) compares the value stored in memory location addr with old and if they are equal, the value

Algorithm 3.4 gProximity balancing kernel - GPU level.

```
1: function balanceWorkList(gStackIn, gStackOut, gTotalNodes)
2:     nodesLeft ← gStackIn.numberOfStacks
3:     iOffset ← thread
4:     sSum[thread] ← 0
5:     while thread < nodesLeft do  ▷ Loop to init sSizes and prepare reduction
6:         stackSize ← gStackIn[iOffset].count
7:         sSizes[iOffset] = stackSize
8:         sSum[thread]+ = stackSize
9:         iOffset+ = sNThreads
10:        nodesLeft− = sNThreads
11:    sync
12:    reduce(sSum, thread, sNThreads)              ▷ Result is copied to index 0
13:    nodesLeft ← sSum[0]
14:    nodesPerStack ← ceil(nodesLeft/sNThreads)
15:    assignShared(totalEntries , nodesLeft)    ▷ Output total to global mem
16:    iOffset ← 0 , oOffset ← 0
17:    inStack ← gStackIn.beforeFirst, inStack.count ← 0
18:    for all Stack s ∈ gStackOut do
19:        nodesLocal ← min(nodesLeft, nodesPerStack)
20:        oOffset ← thread
21:        assignShared(s.count , nodesLocal)  ▷ Final counter output per stack
22:        while nodesLocal > 0 do
23:            if inStack.count <= 0 then
24:                inStack ← inStack.next
25:                iOffset ← thread
26:                continue
27:            nodesToWrite ← min(nodesLocal, inStack.count , sNThreads)
28:            if thread < nodesToWrite then
29:                s[oOffset] = inStack[iOffset]
30:            nodesLocal− = nodesToWrite
31:            iOffset+ = nodesToWrite
32:            oOffset+ = nodesToWrite
33:            inStack.count− = nodesToWrite
34:            nodesLeft− = nodesToWrite
35:            sync
36: end function
```

Algorithm 3.5 ARORA *et al.* (1998) task stealing algorithm

```
1:  assignedThread ← NULL
2:  if self = processZero then
3:      assignedThread ← rootThread          ▷ Assign root thread to process zero
4:  while computationIsNotDone do                                  ▷ Scheduling loop
5:      while assignedThread ≠ NULL do
6:          DISPATCH(assignedThread)          ▷ Execute until terminate or block
7:          assignedThread ← self.deque POPBOTTOM                  ▷ Get next thread
8:      YIELD                                    ▷ Before steal, yield processor
9:      victim ← randomProcess                ▷ Select victim process at random
10:     assignedThread ← victim.deque. POPTOP                      ▷ Try to steal thread
```

stored in memory location addr is swapped with new. In this case, we say the cas succeeds. Otherwise, it loads the value stored in memory location addr into new, without modifying the memory location addr. In this case, we say the cas fails (ARORA *et al.* (1998)).

In the scope of the algorithm, CAS instructions serve to verify if a thread complete reading a deque pointer and changing it without interference from other threads. This is assured by reading a *pointer* to *old*, saving changes on *new* and doing a *cas(pointer, old, new)*. If the *cas* succeeds, then the change on pointer also succeeds. Otherwise, the *cas* fails and the pointer remains unchanged, which means that the whole operation is aborted. Note that this way the inner functions assure that exactly one thread can do the pointer modification at a time.

Another detail in the inner functions is about resetting the queue pointers when a deque becomes empty. This occurs only in *popBottom()*, since *popTop()* is not allowed to empty a deque (see Algorithm 3.6, function *popTop()*, line 3). Resetting means changing both *pop* and *top* pointers to 0 and this operation can lead to race conditions. For example, suppose that a process i is preempted in *popTop()* after stealing a thread from process j (Algorithm 3.6, function *popTop()*, line 6) but before doing the *cas* verification (Algorithm 3.6, function *popTop()*, line 8). Other operations can empty deque d_j, leading to d_j pointers resetting. In this case, another set of operations can restore d_j pointers to their original state. When process i is resumed the cas verification will wrongly succeed since the acquired thread is outdated.

To overcome the aforementioned problem, the pointer *age* is defined to replace the usual *top* pointer on deques. *age* has two fields: the usual *top* pointer and a *tag*. In practice, *tag* is implemented by adapting the "bounded tags" algorithm (MOIR (1997)) as a wraparound counter which is incremented every time *top* is changed. This way even if the set of deque operations mentioned in the above example can lead to equal *top* values, *tags* will be different and the final *cas* verification will fail.

Algorithm 3.6 ARORA *et al.* (1998) task stealing inner functions

```
1: function pushBottom(Thread * thr)
2:     local ← bot
3:     deque[localBot] ← thr
4:     localBot + +
5:     bot ← localBot
6: end function

1: function popTop
2:     oldAge ← age
3:     localBot ← bot
4:     if localBot ≤ oldAge.top then
5:         returnNULL
6:     thr ← deq[oldAge.top]
7:     newAge ← oldAge
8:     newAge.top + +
9:     cas(age, oldAge, newAge)
10:    if oldAge = newAge then
11:        return thr
12:    return NULL
13: end function

1: function popBottom
2:     localBot ← bot
3:     if localBot = 0 then
4:         returnNULL
5:     localBot − −
6:     bot ← localBot
7:     thr ← deq[localBot]
8:     oldAge ← age
9:     if localBot > oldAge.top then
10:        return thr
11:    bot ← 0
12:    newAge.top ← 0
13:    newAge.tag ← oldAge.tag + 1
14:    if localBot = oldAge.top then
15:        cas(age, oldAge, newAge)
16:        if oldAge = newAge then return thr
17:    age ← newAge
18:    return NULL
19: end function
```

Also, *age* is implemented as an integral type to suit the *cas* function and its fields are written and read using bitwise operations. The number of bits of the *tag* field must be chosen wisely to avoid premature wraparound, which can lead to the same problem that it is trying to address. Figure 3.2 shows the deque schematic with all pointers and fields.

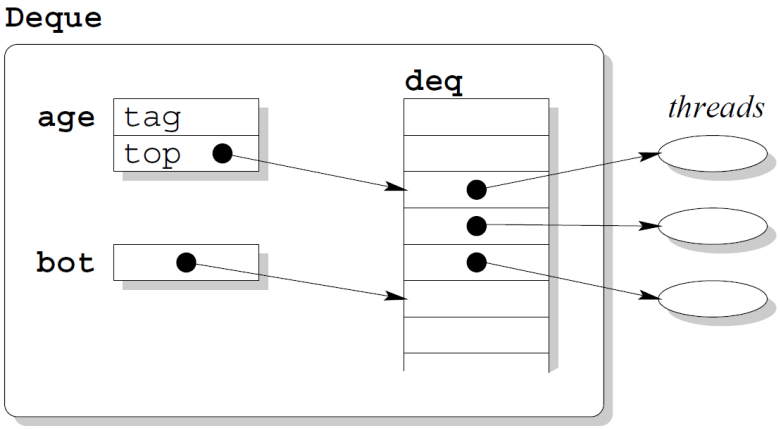

Figure 3.2: ARORA *et al.* (1998) deque representation.

The inner functions algorithms are depicted in Algorithm 3.6. As stated before, it is important to note that just *age* changes can lead to problems in parallelism and therefore all modifications on it are followed by a *cas* call. So *pushBottom()* is very straight-forward, since it just modifies the *bot* pointer. *popTop()* reads *age* and *bot* and verifies if the operations did not empty the deque. If it does, the operation returns a NULL thread since we do not want this operation to empty deques. Otherwise, a thread is assigned for return, *top* is incremented and the *cas* verification is performed. *popBottom()* is also very straight forward. Until line 9, bot is changed and a thread is assigned for return if the deque is not empty. Otherwise, from line 10 on deque pointers are reset and a thread is returned if the cas succeeds.

3.3 Lazy Work Stealing Algorithm for Load Balancing on GPU

The main objective of this novel algorithm is to be more flexible than other methods by diminishing the overhead of load balancing and thus freeing GPU to do the actual work, i.e., the traversal itself. We identified some optimization possibilities in the gProximity approach of LAUTERBACH *et al.* (2010) which will guide our novel algorithm.

1. Working blocks may halt. The idea of totally halting a block working on traversal in favor of others that are idle does not seem reasonable. This is

confirmed by the experiment results of CEDERMAN and TSIGAS (2009). Ideally, a block should be able to get more traversal nodes with a minimal performance hit on other working blocks.

2. Number of blocks: the predefined number of blocks launched for traversal is crucial to the algorithm performance. If it is too low, the balancing kernel is called too often and a great amount of GPU time is used on balancing instead of traversal. Also a lot of CPU-GPU context change is done, which we want to avoid. Ideally, we want to minimize the balancing calls and context changes.

3. Available shared memory per block: the amount of shared memory per block is also ruled by the number of blocks. If we have a large number of blocks launched in a traversal, the amount of shared memory is diminished per block and, if the stacks are implemented in shared memory, a block is more susceptible to halting by full stack. Ideally, we should be able to launch a minimum of blocks, increasing shared memory per block and reducing the number of halts by full stack in case of a shared memory stack implementation. An alternative is implementing traversal without shared memory, using just global memory. But this leads to lesser coalescent global memory accesses, since shared memory can be used to organize data before pushing it to global memory (NVIDIA (2012)). In addition, front decomposition TANG *et al.* (2010b), which is a method commonly used to exploit temporal coherence and accelerate hierarchy traversal can benefit greatly from using shared memory, since the method is memory bound in essence.

The idea of adapting ARORA *et al.* (1998) task stealing algorithm for GPU presented in CEDERMAN and TSIGAS (2009) can be used to explore these problems on static list load balancing.

The problem of working blocks halting is almost totally addressed by using this algorithm, since given two blocks $b0$ and $b1$, $b0$ can call *pushBottom()* or *popBottom()* on its deque at the same time that $b1$ can call *popTop()* on $b0$ deque to steal nodes, unless a pointer reset must be done. Some of the problems related with number of blocks and shared memory are also solved by setting the number of launched blocks for the traversal kernel as a number of blocks that can actually run in parallel on the device and give best device occupancy. This way we achieve maximum shared memory per block and no resource waste since all multiprocessors will be busy. It is important to note that even if a kernel can be launched with a huge number of blocks, just a few reside at the same time on the device and this information can be queried on all current CUDA enabled devices.

Unfortunately not all problems are solved using this approach. The algorithm requires that all blocks with empty deques keep pooling another block deques in a

round robin fashion attempting to steal nodes. In this case even if the multiprocessor is busy it may not actually be doing useful work. Thus, we propose an improved method by performing slight modifications to better suit the stealing part of the algorithm to GPUs.

1. A three-pass approach is used for node management since the amount of shared memory per block can be augmented. In each traversal loop iteration each thread pops one node, evaluates it and all generated work nodes or front nodes are saved in local thread memory (pass 1). Continuing in the same loop iteration, all nodes generated in a block are saved in a shared memory stack. This is achieved by using a prefix-sum (SENGUPTA *et al.* (2007)) approach (pass 2). Ending the iteration, part of the shared stack is pushed to global memory if the shared stack is near full (pass 3).

2. In the pop pass (pass 1), a three-level pop approach is used. First pop attempt is on the shared stack (level 1). If not successful, an attempt to pop from the block's deque is done (level 2). If it fails, work is stolen from the deques of other blocks (level 3) and the block resumes traversal. If no work can be stolen, the traversal ends.

3. The operations in all deques are done in batch to avoid excessive increments on global memory deque pointers. Just one thread in a block is responsible to do the actual operation on deque pointers. The other threads just help to push or pop the affected nodes.

4. A lazy approach is used to acquire stolen nodes (pop level 3). Since a block has a reasonable number of threads running in parallel, the stealing part of the algorithm calls *popTop()* on all deques at the same time, each thread changing the age pointer from one global deque. However, just the nodes from the first stolen deque are actually transferred to the block stack at this time. The transfer is done in little batches to ensure saving shared stack size for traversal. To achieve this lazy transfer, it is necessary to save the deque indices and sizes of all successful *popTop()* operations of a task steal and not letting *popBottom()* operations reset pointers. It is important to note that these changes aim at maximizing stolen nodes in a stealing attempt and maximizing stack size available for future traversal. Specifically, if a block b_i successfully calls *popTop()* on blocks b_j and b_k, saves stealing info (indices and sizes) to p_{ij} and p_{ik} and $j < k$, then nodes from global deque d_k are transferred to local stack s_i only after all global deque d_j nodes are pushed, processed and s_i as well as global deque d_i become empty again.

Figure 3.3 shows the overall schematic of the algorithm. The traversal with front generation (TANG *et al.* (2010b)) is detailed in Algorithms 3.7 to 3.13. The traversal algorithm can be used to perform full hierarchy traversal, i.e., from root node, or from work generated in a front update pass before traversal (TANG *et al.* (2010b)). It is assumed that all variables declared volatile can bypass incoherent cache in memory operations and that all non declared variables have local scope. Also, all SYNC calls are supposed to do the synchronization of threads inside a block.

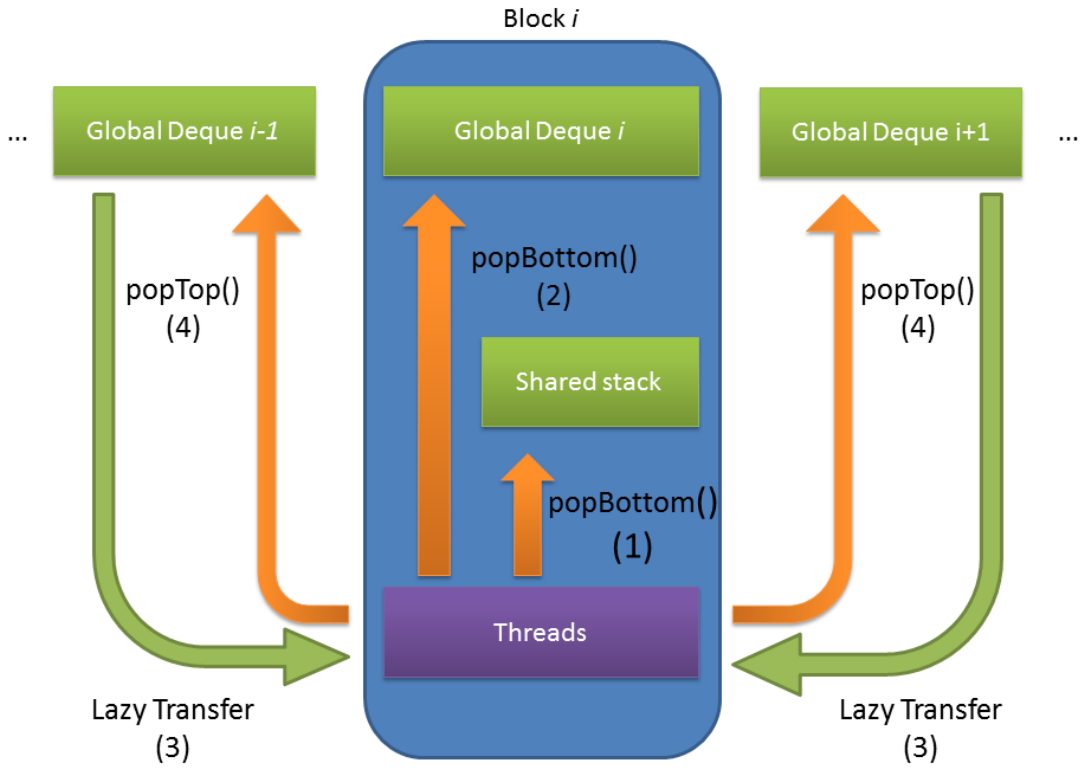

Figure 3.3: Lazy work acquisition scheme. First, threads try to pop from their block's shared stack (1). If not successful, they try to pop from its global deque (2). If not successful, enough nodes (if any) are transferred from lazily stolen nodes (3). If there are no more nodes from lazy steal, all other block deques are stolen (4).

Algorithm 3.7 Novel algorithm traversal and front acquisition

```
1: function TRAVERSE(gBvtt , gFront)
2:     Parameters (global memory):
3:         gBvtt                                          ▷ BVTT work deques
4:         gFront                                         ▷ BVTT front deques
5:     Block shared memory:
6:         sBvtt              ▷ BVTT work shared: stack and prefix-sum data
7:         sFront             ▷ BVTT front shared: stack and prefix-sum data
8:     Thread local memory variables:
9:         tBvtt[3]                                       ▷ generated work nodes
10:        tFront                                         ▷ generated front node
11:        thread          ▷ Thread index. Declared once here for all algorithms.
12:        block           ▷ Block index. Declared once here for all algorithms.
13:    loop
14:        if NOT(popWork(gBvtt , sBvtt , tBvtt)) then                ▷ Pass 1
15:            pushRemainingFront
16:            return TRAVERSAL_END
17:        if NOT(tBvtt.empty) then
18:            evaluateCollision(tBvtt , tFront)
19:        compactStacks(sBvtt , tBvtt , sFront , tFront)             ▷ Pass 2
20:        if NOT(pushWork(gBvtt , sBvtt) then          ▷ Pass 3: BVTT work
21:            return OVERFLOW
22:        if NOT(pushWork(gFront , sFront) then        ▷ Pass 3: BVTT front
23:            return OVERFLOW
24:    end loop
25: end function
```

Algorithm 3.8 Shared stacks compaction related functions. compactStacks receives both bvtt and front items generated in the current traversal iteration separated per thread. It compacts these items and push them into the stack.

```
1: function compactStacks(sBvtt , tBvtt , sFront , tFront)
2:     sBvtt.threadCounters[thread] ← tBvtt.count
3:     sFront.threadCounters[thread] ← tFront.count
4:     sync
5:     prefixSumCompaction(sBvtt , tBvtt)
6:     prefixSumCompaction(sFront , tFront)
7:     sync
8:     finalBvttBot ← sBvtt.stack.bottom + sBvtt.totalTNodes
9:     finalFrontBot ← sFront.stack.bottom + sBvtt.totalTNodes
10:    assignShared(sBvtt.stack.bottom, finalBvttBot)
11:    assignShared(sFront.stack.bottom, finalFrontBot)
12:    sync
13: end function

1: function prefixSumCompaction(sData , tItems)
2:     offset ←prefixSum(sData.threadCounters , thread , tItems.count)
3:     for i = 0 → tItems.count do
4:         stack = sData.stack
5:         stack[stack.bottom + offset + i] ← tItems[i]
6:     if thread = lastThread then   ▷ Last thread calc the total from prefix-sum
7:         sData.totalTNodes ← offset + tItems.count
8: end function
```

Algorithm 3.9 Novel lazy steal popWork device function. Each thread tries to pop a node in a 3 level approach until successful or all deques are inactive.

```
1: function popWork(gBvtt , sBvtt , tBvtt)
2:     Block shared memory:
3:         pFlag[3]                                   ▷ Pop results for each level pass
4:         pStrtIdx[NDEQUES]                          ▷ Saved pop start indices
5:         pSizes[NDEQUES]                            ▷ Saved pop sizes
6:         victim                                     ▷ Current work stealing victim
7:         nDeques      ▷ Number of deques. Declared once here for all algorithms
8:     if thread = firstThread then                   ▷ Level 1, pop from shared stack
9:         stack ← sBvtt.stack
10:        pFlag[L1] ← stack.popBottom(pStrtIdx[block] , pSizes[block])
11:        pFlag[L3] ← bPopFlag[L1]
12:    Sync
13:    if pFlag[L1] then
14:        if thread < pSizes[block] then
15:            tBvtt ← sBvtt.stack[pStrtIdx[block] + thread]
16:    else
17:        level2Pop(gBvtt , pFlag , pStrtIdx , pSizes , victim)
18:        sync
19:        if NOT(pFlag[L2]) then                     ▷ Work stealing
20:            level3Pop(gBvtt , pFlag , pStrtIdx , pSizes , victim)
21:        sync
22:        if pFlag[L3] AND thread < pSizes[block] then
23:            tBvtt ← gBvtt.deque[victim][pStrtIdx[block] + thread]
24:    return pFlag[L3]
25: end function
```

Algorithm 3.10 Level 2 pop function. Pop from the deque owned by the block.

```
1: function level2Pop(gBvtt , pFlag , pStrtIdx , pSizes , victim)
2:     Global memory:
3:         gBvtt.active[] is volatile
4:     if thread = firstThread then
5:         deque ← gBvtt.deque[block]
6:         pFlag[L2] ← deque.popBottom(pStrtIdx[block] , pSizes[block])
7:         if pFlag[L2] then
8:             victim ← block
9:         else
10:            gBvtt.active[block] ← INVALID          ▷ Not eligible as victim
11:            pStrtIdx[block] ← ∞
12:            victim ← ∞
13:        pFlag[L3] ← pFlag[L2]
14: end function
```

Algorithm 3.11 Level 3 pop function. Lazy work steal.

```
1: function LEVEL3POP(gBvtt , pFlag , pStrtIdx , pSizes , victim)
2:     Global memory:
3:         gBvtt.active[] is volatile
4:     Block shared memory:
5:         activeDequesFlag                          ▷ Mark if there are active blocks.
6:     if thread < nDeques then                      ▷ Lazy node acquisition
7:         if pStrtIdx[thread] ≠ INVALID then
8:             victim ← ATOMICMIN(victim , thread)
9:             pFlag[L3] ← TRUE
10:    SYNC
11:    if thread = victim then
12:        pStrtIdx[block] ← pStrtIdx[thread]
13:        pSizes[block] ← pSizes[thread]
14:        pStrtIdx[thread] ← INVALID ▷ Mark lazy steal as done for the victim
15:    while NOT(pFlag[L3]) do                       ▷ No more work for lazy steal.
16:        SYNC
17:        ASSIGNSHARED(activeDequesFlag , FALSE) ▷ No active deques found
18:        SYNC
19:        if thread < nDeques then
20:            if gBvtt.active[thread] = TRUE then
21:                activeDequesFlag ← TRUE           ▷ Found an active deque
22:                deque ← gBvtt.deque[thread]
23:                if deque.POPTOP(pStrtIdx[thread] , pSizes[thread]) then
24:                    gBvtt.active[block] ← TRUE                ▷ Work found
25:                    victim ← ATOMICMIN(victim , thread)
26:                    pFlag[L3] ← TRUE
27:        SYNC
28:        if activeDequesFlag = FALSE then          ▷ All deques inactive.
29:            break
30:        if thread = victim then                   ▷ Lazy steal the first found deque
31:            pStrtIdx[block] ← pStrtIdx[thread]
32:            pSizes[block] ← pSizes[thread]
33:            pStrtIdx[thread] ← INVALID
34:        SYNC
35: end function
```

Algorithm 3.12 Novel algorithm pushWork device function. Pushes data to global memory if the number of nodes in the shared stack is greater than a predetermined threshold

```
1: function pushNodes(gBvtt , sBvtt)
2:     bottom ← sBvtt.stack.bottom
3:     if bottom ≥ THRESHOLD then
4:         NPushLoops ← bottom/nThreads
5:         for i = 0 → NPushLoops do
6:             stackOffset ← bottom − (i + 1) ∗ nThreads + thread
7:             dequeOffset ← gBvtt.deque[block].bottom
8:             dequeOffset+ = i ∗ NTHREADS + thread
9:             gBvtt.deque[block][dequeOffset] ← sBvtt.stack[stackOffset]
10:        sync
11:        if thread = firstThread then
12:            gBvtt.deque[block].bottom+ = NPushLoops ∗ nThreads
13:            sBvtt.stack.bottom− = NPushLoops ∗ nThreads
14:        memoryFence                                   ▷ Cache coherence
15: end function
```

Algorithm 3.13 Novel algorithm inner deque/stack device functions. Change deque pointers

```
1: function DEQUE.POPTOP(pStrtIdx , pSize)
2:     Global memory:
3:         top is volatile
4:         bottom is volatile
5:     oldTop ← top
6:     localBot ← bottom
7:     if localBot − oldTop < 2 ∗ POP_SIZE + 1 then
8:         return FAIL                    ▷ popTop is not allowed to empty the list
9:     size ← MIN(POP_SIZE , localBot - oldTop)
10:    newTop = oldTop + size
11:    ATOMICCAS(top , oldTop , newTop)
12:    if oldTop = newTop then
13:        pStrtIdx ← index
14:        pSize ← size
15:        return SUCESSFUL
16:    else
17:        return FAIL
18: end function

1: function DEQUE.POPBOTTOM(pStrtIdx , pSize)
2:     Global memory:
3:         top is volatile
4:         bottom is volatile
5:     localTop ← top
6:     localBot ← bottom
7:     if localBot = localTop then
8:         return FAIL
9:     localBot− = POP_SIZE
10:    if localBot > localTop then
11:        pSize ← POP_SIZE
12:        pStrtIdx ← localBot
13:        bottom ← localBot
14:    else                    ▷ Fix pStrtIdx, pSize and bottom if bottom pass top
15:        pSize ← localBot + POP_SIZE − localTop
16:        pStrtIdx ← localTop
17:        bottom ← localTop
18:    return SUCESSFUL
19: end function
```

Chapter 4

Implementation Details and Experiments

This chapter describes the implementation details of the proposed algorithm, mainly the concerns about memory coherence and inter-block communication. Also the experimental methodology and performance results used to evaluate the algorithm are presented. Finally the method is compared with the load balancing approach used in gProximity LAUTERBACH *et al.* (2010).

4.1 Implementation Details

The algorithm was implemented using NVIDIA CUDA technology (NVIDIA (2012)). As discussed in Section 2.1, CUDA architecture is based on a memory hierarchy composed of a high latency abundant global memory and an on-chip low latency scarce shared memory. To achieve better performance on global memory accesses, two levels of low latency cache memory are used. Figure 4.1 shows the cache memory hierarchy for global memory. The green boxes represent the device memory and the blue box represents the system memory. The GPU Memory box represents the device's global memory and above it the cache memory levels L2 and L1. It is important to remark that access to L2 is coherent, while L1 is incoherent. This means that a change in global memory done by a SM is not assured to be seen by another SM. This aspect forbids design of algorithms using inter-block communication, since blocks can run in any SM.

To overcome this problem, CUDA provides programmers with some options. The first is compiling code with L1 cache deactivated using flag *-Xptxas -dlcm=cg*. This solution has an overall performance penalty since all code is affected, even parts that could benefit using L1 cache. The second is the usage of atomic operations which by definition propagates changes to higher level cache. The third is coding using the

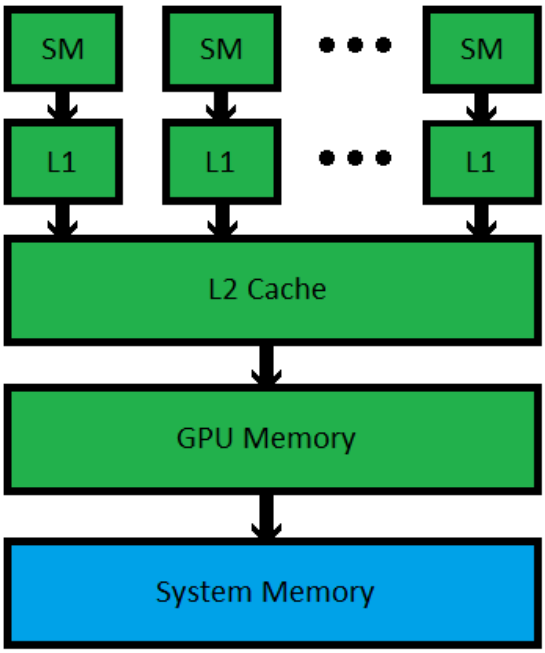

Figure 4.1: NVIDIA CUDA cache hierarchy

volatile keyword on specific data pointers. This keyword tells the compiler to bypass L1 cache in memory accesses related with that pointer. Another option is to use *_threadfence()* calls to force threads to propagate write accesses to upper level cache. The choice of how to use these tools depends on some hardware aspects, such as the code target architecture. For instance, atomic operations have better performance on newer architectures. As can be seen in Algorithms 3.7 to 3.13 the design choice was to use a mix of atomic operations, *volatile* keyword and *_threadfence()*.

4.2 Experiments

The algorithm was tested in two systems. The first uses a GeForce GTS 450 card, which has 57.7 GB/s memory bandwidth and 4 SMs (streaming multiprocessors), each one with 48 CUDA cores, resulting in a total of 192 CUDA cores. The second uses a GeForce GT 520 card, which has 14.4 GB/s memory bandwidth and 1 SM, resulting in a total of 48 CUDA cores. Several commonly benchmarks were used, namely the BART (BAR (2013)), Funnel, Cloth/Ball and N-body models (GOVINDARAJU *et al.* (2013)). The BART benchmark evaluates inter-collisions, and we have used four different resolutions (64, 256 , 1024 and 4096 triangles). The Funnel benchmark has 18.5K triangles and tests self-collision in deformable motion. Cloth/Ball benchmark evaluates the same type of collision, but has a heavier workload with 92K triangles. The N-body benchmark has 146K triangles and mainly tests collision in rigid body motion; this benchmark was tested only in the GeForce

GT520, since the memory footprint was prohibitive for the other system.

Table 4.1 shows the performance results for each benchmark. The numbers include time for hierarchy refit, front update, front-based traversal, load balancing and triangle pair intersection. The implementation uses the non-penetration filters TANG *et al.* (2010a) and orphan sets TANG *et al.* (2008) culling methods. Table 4.2 shows the ratio of the GTS 450 and the GT520 timings and demonstrates how the algorithm scales.

Table 4.3 shows the average number of processed front nodes per frame. This information depicts well the amount of resources necessary to compute CCD for high density models and the differences in timings among benchmarks.

Table 4.1: Lazy Work Stealing performance results. Times in ms.

Model	Triangles	GTS 450	GT 520
BART64	64	2.6	1.6
BART256	256	3.4	3.5
BART1024	1024	8.4	19.7
BART4096	4096	51.3	169.9
Funnel	18.5K	17.4	48.2
Cloth/Ball	92K	77.3	225.7
N-body	146K	-	1172.2

Table 4.2: Lazy Work Stealing speedup.

Model	Triangles	Speedup
BART64	64	0.61
BART256	256	1.03
BART1024	1024	2.34
BART4096	4096	3.31
Funnel	18.5K	2.77
Cloth/Ball	92K	2.92

Table 4.3: Lazy Work Stealing average number of processed front nodes per frame.

Model	Front nodes
BART64	807.996
BART256	9.15953K
BART1024	127.394K
BART4096	2.15626M
Funnel	493.391K
Cloth/Ball	2.48204M
N-body	18.3137M

4.3 Comparison and Analysis

The Lazy Work Stealing algorithm is compared with gProximity LAUTERBACH *et al.* (2010) in an implementation done by the author of this paper. Table 4.4 shows the performance timings of the gProximity implementation for all systems and benchmarks, whereas Table 4.5 indicates the scalability of the algorithm with respect to the resources available in both systems.

It is important to note that the Table 4.1 and Table 4.4 have numbers generated with similar code for refit and triangle intersection. The only noticeable changes are in the load balancing, front update and traversal code. Also, this implementation uses the same culling methods used in the Lazy Work Stealing implementation. Figure 4.2 shows how the processing time is distributed in the BVH management for the novel Lazy Work Stealing algorithm.

Table 4.4: gProximity performance results. Times in ms.

Model	Triangles	GTS 450	GT 520
BART64	64	3.1	1.6
BART256	256	3.6	3.1
BART1024	1024	7.5	14.7
BART4096	4096	47.6	173.4
Funnel	18.5K	16.3	49.0
Cloth/Ball	92K	68.6	238.3
N-body	146K	-	1269.5

Table 4.5: gProximity speedup.

Model	Triangles	Speedup
BART64	64	0.51
BART256	256	1.16
BART1024	1024	1.96
BART4096	4096	3.64
Funnel	18.5K	3.00
Cloth/Ball	92K	3.47

Based on the collected data, the algorithms seem to have similar performance for the analyzed benchmarks. Which algorithm performs better depends on the benchmark. Both algorithms seem to scale well. The GTS 450 system has 4 times more resources than the GT520 system and and both systems have a near 3 times speedup, with gProximity having a peak of 3.47 speedup for the Cloth/Ball benchmark. The BART64 and BART256 examples showed poor scalability because of the lack of work available. Also it was observed that most of the time for BVH management is spent with the front update. This is mainly because of the number

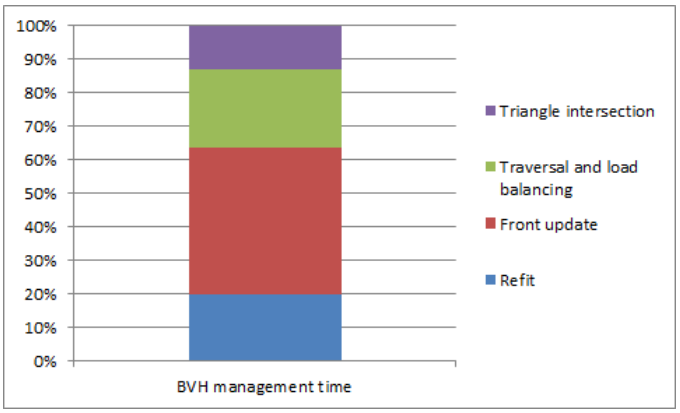

Figure 4.2: Lazy Work Stealing time chart for the Cloth/Ball benchmark.

of nodes that a front can have. For instance, the peaks in the number of processed front nodes pass 10 million for the Cloth/Ball benchmark and 22 million for the N-body benchmark.

Chapter 5

Conclusion

In this work, the Lazy Work Stealing algorithm for load balance of continuous collision detection on GPUs is presented. The algorithm relies on heavy usage of device shared memory to diminish the overhead of node management on traversal. Also, it tries to diminish work acquisition overhead using a greedy steal, lazy transfer approach. This chapter is a report of the algorithm limitations, some directions for future work and further conclusions.

5.1 Limitations

The proposed algorithm has some limitations. First, it is highly memory bound. Thus, the performance is very dependent of the device memory bandwidth, mainly because of the front update pass. Second, the best size of the shared stacks depends on the model. If the model has a lighter workload, setting a higher stack size can forbid nodes to get to the global deques and in consequence forbid blocks to steal work. Analogously, if the model has a heavier workload, setting a lesser stack size can forbid blocks to benefit from the performance of the shared memory latency. Finally, the requirements of global memory size are higher when compared with methods that use deques that can be reset.

5.2 Future Work

The immediate plans for future work include the implementation of more culling methods, such as Representative Triangles CURTIS *et al.* (2008) and Continuous Normal Cones TANG *et al.* (2008), to achieve better performance. Another path of development is to test the algorithm with more modern GPUs, such as NVidia's Kepler compute architecture.

In addition, a more deep comparison of Lazy Work Stealing with other load

balancing approaches could be an interesting topic as well as the usage of this load balancing algorithm on other problems where work loads depend on the geometry, such as Ray Tracing.

Bibliography

2013, "Benchmark for Animated Ray Tracing", Available at: <http://www.ce.chalmers.se/research/group/graphics/BART/>.

AJMERA, P., GORADIA, R., CHANDRAN, S., et al., 2008, "Fast, parallel, GPU-based construction of space filling curves and octrees". In: *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pp. 10:1–10:1, New York, NY, USA. ACM. ISBN: 978-1-59593-983-8. doi: 10.1145/1342250.1357022. Available at: <http://doi.acm.org/10.1145/1342250.1357022>.

ARORA, N. S., BLUMOFE, R. D., PLAXTON, C. G., 1998, "Thread scheduling for multiprogrammed multiprocessors". In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pp. 119–129, New York, NY, USA. ACM. ISBN: 0-89791-989-0. doi: 10.1145/277651.277678. Available at: <http://doi.acm.org/10.1145/277651.277678>.

CEDERMAN, D., TSIGAS, P., 2009, "On sorting and load balancing on GPUs", *SIGARCH Comput. Archit. News*, v. 36, n. 5 (jun.), pp. 11–18. ISSN: 0163-5964. doi: 10.1145/1556444.1556447. Available at: <http://doi.acm.org/10.1145/1556444.1556447>.

CURTIS, S., TAMSTORF, R., MANOCHA, D., 2008, "Fast collision detection for deformable models using representative-triangles". In: Haines, E., McGuire, M. (Eds.), *Proceedings of the 2008 Symposium on Interactive 3D Graphics, SI3D 2008, February 15-17, 2008, Redwood City, CA, USA*, pp. 61–69. ACM. ISBN: 978-1-59593-983-8. doi: http://doi.acm.org/10.1145/1342250.1342260.

ERICSON, C., 2004, *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. ISBN: 1558607323.

GOTTSCHALK, S., LIN, M. C., MANOCHA, D., 1996, "OBBTree: a hierarchical structure for rapid interference detection". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pp. 171–180, New York, NY, USA. ACM. ISBN: 0-89791-746-4. doi: 10.1145/237170.237244. Available at: <http://doi.acm.org/10.1145/237170.237244>.

GOVINDARAJU, N., KABUL, I., REDON, S., et al., 2013, "UNC Dynamic Scene Benchmarks", Available at: <http://gamma.cs.unc.edu/DYNAMICB/>.

GROUP, K., 2012. "The OpenCL Specification Version: 1.2 Document Revision: 19". Available at: <http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>.

HEO, J.-P., SEONG, J.-K., KIM, D., et al., 2010, "FASTCD: Fracturing-Aware Stable Collision Detection". In: Popovic, Z., Otaduy, M. A. (Eds.), *Proceedings of the 2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, SCA 2010, Madrid, Spain, 2010*, pp. 149–158. Eurographics Association. ISBN: 978-3-905674-27-9. doi: http://dx.doi.org/10.2312/SCA/SCA10/149-158.

HERLIHY, M., 1991, "Wait-free synchronization", *ACM Trans. Program. Lang. Syst.*, v. 13, n. 1 (jan.), pp. 124–149. ISSN: 0164-0925. doi: 10.1145/114005.102808. Available at: <http://doi.acm.org/10.1145/114005.102808>.

KIM, D., HEO, J.-P., HUH, J., et al., 2009, "HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs", Computer Graphics Forum (Pacific Graphics). Available at: <http://sglab.kaist.ac.kr/HPCCD/>.

LAUTERBACH, C., GARL, M., SENGUPTA, S., et al., 2009, "Fast bvh construction on gpus". In: *In Proc. Eurographics '09*.

LAUTERBACH, C., MO, Q., MANOCHA, D., 2010, "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries". In: *Proceeding of Eurographics 2010*.

MOIR, M., 1997, "Practical implementations of non-blocking synchronization primitives". In: *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pp. 219–228, New York, NY, USA. ACM. ISBN: 0-89791-952-1. doi: 10.1145/259380.259442. Available at: <http://doi.acm.org/10.1145/259380.259442>.

NGUYEN, H., 2007, *Gpu gems 3*. Addison-Wesley Professional. ISBN: 9780321545428.

NI, T., 2009. "DirectCompute". Available at: <http://www.gputechconf.com/object/gtc2009-on-demand.html#session1015>.

NICKOLLS, J., DALLY, W. J., 2010, "The GPU Computing Era", *IEEE Micro*, v. 30, n. 2 (mar.), pp. 56–69. ISSN: 0272-1732. doi: 10.1109/MM.2010.41. Available at: <http://dx.doi.org/10.1109/MM.2010.41>.

NICKOLLS, J., BUCK, I., GARLAND, M., et al., 2008, "Scalable Parallel Programming with CUDA", *Queue*, v. 6, n. 2 (mar.), pp. 40–53. ISSN: 1542-7730. doi: 10.1145/1365490.1365500. Available at: <http://doi.acm.org/10.1145/1365490.1365500>.

NVIDIA, 2012, *NVIDIA CUDA Programming Guide*. Available at: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., et al., 2007, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3 ed. New York, NY, USA, Cambridge University Press. ISBN: 0521880688, 9780521880688.

PROVOT, X., 1997, "Collision and self-collision handling in cloth model dedicated to design garments". In: *Graphics Interface 97*, pp. 177–179.

SENGUPTA, S., HARRIS, M., ZHANG, Y., et al., 2007, "Scan primitives for GPU computing". In: *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pp. 97–106, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. ISBN: 978-1-59593-625-7. Available at: <http://dl.acm.org/citation.cfm?id=1280094.1280110>.

TANG, M., CURTIS, S., YOON, S.-E., et al., 2008, "Interactive continuous collision detection between deformable models using connectivity-based culling". In: *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pp. 25–36, New York, NY, USA. ACM. ISBN: 978-1-60558-106-2. doi: http://doi.acm.org/10.1145/1364901.1364908.

TANG, M., MANOCHA, D., TONG, R., 2010a, "Fast continuous collision detection using deforming non-penetration filters". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*,

I3D '10, pp. 7–13, New York, NY, USA, a. ACM. ISBN: 978-1-60558-939-8. doi: 10.1145/1730804.1730806. Available at: <http://doi.acm.org/10.1145/1730804.1730806>.

TANG, M., MANOCHA, D., TONG, R., 2010b, "MCCD: Multi-Core collision detection between deformable models using front-based decomposition", *Graphical Models*, v. 72, n. 2, pp. 7–23. ISSN: 1524-0703. doi: DOI: 10.1016/j.gmod.2010.01.001.

TESCHNER, M., KIMMERLE, S., ZACHMANN, G., et al., 2004. "Collision Detection for Deformable Objects". Available at: <http://www-evasion.imag.fr/Publications/2004/TKZHRFCFMS04>.