Juha Koivistoinen

An investigation of Signed-Volume Gilbert-Johnson-Keerthi algorithm in collision detection.

Master's Thesis in Information Technology

December 9, 2019

University of Jyväskylä

Faculty of Information Technology

Author: Juha Koivistoinen

Contact information: juha.t.koivistoinen@gmail.com

Supervisor: Paavo Nieminen, Tuomo Rossi

Title: An investigation of Signed-Volume Gilbert-Johnson-Keerthi algorithm in collision detection.

Työn nimi: Signed-Volume Gilbert-Johnson-Keerthi algoritmin tutkimus törmäystarkasteluissa.

Project: Master's Thesis

Study line: Games and gamification

Page count: 60+0

Abstract: In this thesis, an investigation of the Signed-Volume Gilbert-Johnson-Keerthi collision detection algorithm is presented. The principles of video game physics engines, the general flow of the collision detection process and the GJK algorithm itself, are reviewed. Additionally, running the algorithm with graphics card, and the relevant related topics, such as GPGPU and CUDA SDK, are introduced. A simulation software was implemented for both CPU and GPU following the presented principles, and some experiments were conducted. The results acquired from the tests are discussed. In addition, some self-reflecting notions and discussion about implementing a simulation software for similar experiments, are brought up in later chapters.

Keywords: GPU, CPU, GJK, GPGPU, Collision detection, spatial partition, parallel thinking, physics engine, game engine, video game physics

Suomenkielinen tiivistelmä: Tässä tutkielmassa esitellään Signed-Volume Gilbert-Johnson-Keerthi törmäystarkastelu-algoritmi. Videopelien fysiikkamoottoreiden, törmäystarkatelun yleinen kulku ja GJK - algoritmi itsessään käydään läpi oleellisimmilta osin. Työssä paneudutaan myös esitellyn algoritmin suorittamiseen grafiikkasuorittimella, ja esitellään siihen liittyvät tärkeimmät aiheet, kuten GPGPU ja CUDA SDK. Tutkielmaa varten tehtiin kokeita, joita varten implementoitiin esitettyjen periaatteiden mukainen simulaatio-ohjelmisto

CPU ja GPU suoritukseen. Näistä kokeista saadut tulokset esitellään ja niistä keskustellaan. Myös reflektoinnin omaisia huomioita ja keskustelua vastaavanlaisen ohjelmiston implementaatiosta käydään myöhemmissä kappalesissa.

Avainsanat: GPU, CPU, GJK, GPGPU, Collision detection, spatial partition, parallel thinking, physics engine, game engine, video game physics

List of Figures

Contents

1 Introduction

The demand for better graphics in video games and technological advancement boosted the manufacturing of 3D graphics cards in mid nineties (Hook 1995). It was around that same time when the term *game engine* surfaced and was made familiar to the public (Gregory 2009). A game engine is basically a complex software, as described by Gregory (2009), consisting of separate components programmed to handle physics, sound, rendering etc. which can be reused. When making a new game, it was not necessary to code all these features from scratch anymore, but to use the components provided by the engine and simply code the new game's logic and interactions, instead. *Physics engine* represents a component of a game engine, responsible of handling physics calculations of the engine, for example, for rigid bodies and their collisions. Millington (2010) states that, ideally, a physics engine is a reusable component that handles the underlying physics simulations of a game, and it is totally independent of the scenario. Today, in terms of gaming experience, physics engine is one of the most crucial aspects of technical implementation of a video game (Ericson 2004). Also, as brought out by Gregory (2009), it is common that video game manufacturers attach third-party physics engine libraries to their own engines, instead of developing their own simulation software.

The role of graphics cards, since their invention has shifted from being a one-purpose, graphics-acceleration processors with build-in functionalities, towards programmable multi-purpose computation units. The first fully programmable modern GPU is often considered to be GeForce 256, which was released by NVIDIA Corporation in 1999 (*NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256* 1999). After the release of the new version (2.0) of graphics API, OpenGL (*OpenGL 2.0: OpenGl version history, Khronos* 2004), it became popular to write shader programs with GLSL, a C based language, giving possibility to write code for GPU. In the late 2007, NVIDIA Corporation released general purpose GPU (GPGPU) API, CUDA, which was followed by Khronos group counterpart, OpenCL (*Khronos Launches Heterogeneous Computing Initiative* 2008). Because of this evolution, GPUs today are also usable for more general purposes, such as scientific calculations (Stone et al. 2007), machine learning (Bergstra et al. 2011), data mining (Böhm et al. 2009) and cryptography (Cheng et al. 2018). Due to this evolution, it also became more feasible harnessing GPU for simulating physics in video games.

In this thesis, a an improved Gilbert-Johnson-Keerthi (GJK) distance algorithm (Montanari, Petrinic, and Barbieri 2017) is conducted by applying it to rigid body simulations. The algorithm improves the original GJK (Gilbert, Johnson, and Keerthi 1988) by fixing the setbacks of Johnson sub-algorithm with Signed-Volume sub-algorithm. The theoretical basis for rigid body simulations, collision detection and GPU programming are reviewed and a simulation software coded for experimentation, and the provided results are discussed. The implementation is coded with C++ for CPU and the GPU implementation is done using CUDA SDK. The reader will also be provided with an overview of evolution and the GPU hardware, functionality and coding convensions are introduced.

A central element in physics engines is the collision detection system, and how collisions are handled, can make or break the game's commercial success. In video games, the original GJK developed by Gilbert, Johnson, and Keerthi (1988) is one of the most utilized collision detection algorithms for the narrow-phase stage of the collision detection process. According to Karras (2012), certain stages of a collision detection process are parallelizable. This makes the utilized graphics processor, GeForce GTX 1060, with more than a thousand CUDA cores, ideal for handling collision detection. According to Montanari, Petrinic, and Barbieri (2017), the improved GJK improves the performance of the conventional GJK by 15 - 30 %, thus indicating an improved player experience, and a possibility for developers to allocate more for other CPU side executions. This, combined with the praised performance of modern low level graphics API could provide a motivation for commercial actors to code CUDA implementations to their engines.

1.1 Formalism

It is convenient to list and describe mathematical formalism here, as the thesis contains a fair amount of mathematical formulae and symbols. This section contains a complete list of mathematical formalism utilized in this thesis. If the reader should have problems understanding notations in the thesis, this section should act as a reminder:

Table 1: Formalism

Name	Description	Notation	Example
scalar	lower case letter	a	$a+b=0$
set	upper case letter	A	$A=\varnothing$
set		$\{...\}$	$i\in\{1,2,3,...,j \mid i\leq\mathbb{N}\}$
vector	lower case bold letter	$\mathbf{v}$	$\mathbf{u}+\mathbf{v}=\mathbf{w}$
matrix	upper case bold letter	$\mathbf{M}$	$\mathbf{A}+\mathbf{B}=\mathbf{C}$
quaternion	lower case bold letter with hat	$\hat{\mathbf{q}}$	$\hat{\mathbf{q}}=(q_w,\mathbf{q}_\mathbf{v})$ $\mathbf{q}_\mathbf{v}\in\mathbb{R}^\mathbf{3}$,$q_w\in\mathbb{R}$
scalar product in N dimensions		$\mathbf{u}\cdot\mathbf{v}$	$\mathbf{w}=\mathbf{u}\cdot\mathbf{v}=\sum_{i=1}^{N}u_iv_i$
cross product		$\mathbf{u}\times\mathbf{v}$	$\mathbf{n}=\mathbf{v}\times\mathbf{u}$
matrix multiplication		$\mathbf{AB}$	$\mathbf{C}=\mathbf{AB}$
matrix transformation		$\mathbf{Aa}$	$\mathbf{a}=\mathbf{Ab}$
function		$f(x)$	$f(x)=5x^2+b$
function with vector variable		$f(\mathbf{x})$	$f(\mathbf{x})=\mathbf{x}\times\mathbf{x}+\mathbf{b}$
vector function		$\mathbf{f}(x)$	$\mathbf{f}(x)=g(x)\mathbf{u}\times\mathbf{v}+h(x)\mathbf{b}$
matrix dependent on x	matrix with one or more elements depending on x	$\mathbf{A}(x)$	
time derivative		$\frac{d}{dt}f(t)$	$\frac{d}{dt}\mathbf{F}(t)=\mathbf{v}(t)m$
time derivative		$\dot{\mathbf{X}}(t)$	$\frac{d}{dt}\mathbf{X}(t)=\dot{\mathbf{X}}(t)$
2nd time derivative		$\ddot{\mathbf{X}}(t)$	$\frac{d}{dt}\dot{\mathbf{X}}(t)=\ddot{\mathbf{X}}(t)$
set size		$\lvert...\rvert$	$\lvert A\rvert\geq\lvert B\rvert$
vector length		$\lVert...\rVert$	$\mathbf{r}\cdot\mathbf{n}:\lVert\mathbf{n}\rVert=1$
and		$\wedge$	$a,b\in A:a>b\wedge b>0$
or		$\vee$	$ab=c:a<0\vee b<0$
so that	mid bar or colon	$\mid$ or $:$	$a<b$: $a<0$

2 Physics simulations in video games

Physics in game engines are handled by a specialized engine, sometimes referred as physics engine. Commonly, physics engine may consist of handling particle effects, fluids and rigid-body mechanics, and includes some type of collision detection system (Gregory 2009). A general flow of physics simulation process done in games is depicted in Figure 1.

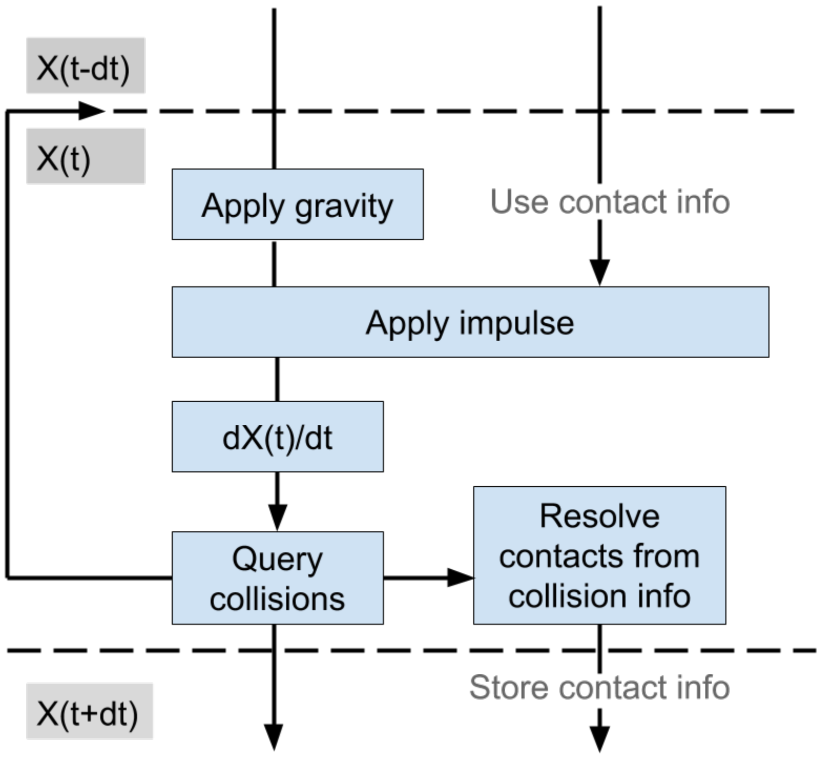

Figure 1. Schematic representation of a typical scenario of retroactive detection. The figure depicts the portion of the rigid body engine that is run on update-phase in game engines. On the left side, the arrows depict the propagation of the rigid body state, and the contact evolution is depicted on right. The system is a representation of the one described by Baraff and Witkin (1997)

Traditionally, in terms of software architecture, physics evaluations are situated in the update - section of the main game-loop (Figure 2), and is thus evaluated after user input or enemy action and before rendering. However, more evolved techniques such as multi-threading and GPU usage may alter or intertwine the exact execution order these tasks.

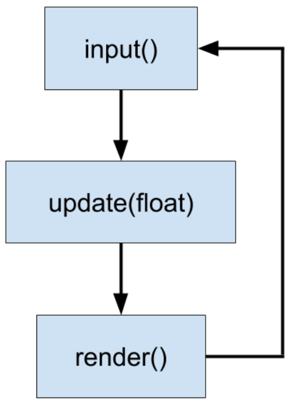

Figure 2. Game loop used in the simulation software.

2.1 Rigid body physics

In the following section, basic building blocks of rigid body physics, needed to perform rigid body physics in computer simulations and games are described. In rigid body simulations, the collisions between bodies are concidered elastic and therefore the kinetic energy is conserved, as other forms of energy such as, heat or potential energy are neglected and also, the bodies are considered to preserve their shapes throughout the simulation. In practice, the inelastic can be approximated by using a retention parameter.

2.1.1 State

The central piece of rigid body simulation is the state vector, which evolves with passing time. The state vector consists of state variables, which govern the physics of the simulation. Thus, these variables store the information between about the positions and momenta of each body between timesteps. They are defind as center of mass dependent variables but they hold no information about the actual vertice positions of the bodies. It is assumed, the center of mass of each body with respect to its vertices remains constant throughtout the simulation.

More accurately, $\mathbf{X}(t)$ is composed of time-dependent state variables: center of mass, orientation, and linear and angular momenta, $\mathbf{x}(t)$, $\mathbf{R}(t)$, $\mathbf{P}(t)$ and $\mathbf{L}(t)$, respectively.

$$\mathbf{X}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{pmatrix} => \frac{d}{dt}\mathbf{X}(t) = \begin{pmatrix} \mathbf{v}(t) \\ \Omega(t)^*\mathbf{R}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{pmatrix} \tag{2.1}$$

$$\frac{d}{dt}\begin{pmatrix} \mathbf{x}(t) \\ \hat{\mathbf{q}}(t) \\ \mathbf{v}(t)/m \\ \omega(t)\mathbf{I}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \frac{\mathbf{1}}{\mathbf{2}}\hat{\omega}(t)\hat{\mathbf{q}}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{pmatrix} \tag{2.2}$$

In the above

$$\omega(t) = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} => \Omega(t)^* = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{2.3}$$

and $\hat{\omega}(t)$ is a quaternion composed of $\omega(t)$, as such:

$$\hat{\omega}(t) = (\omega_w, \omega(t)) : \omega_w = 0 \tag{2.4}$$

As Baraff and Witkin (1997) describe, in this type of rigid body system mass, m and inertia tensor of the body I_{body} are constants. Velocity, $\mathbf{v}(t)$, angular velocity, $\omega(t)$, and inertia tensor $\mathbf{I}(t)$ (or $\mathbf{I}^{-\mathbf{1}}(t)$) are treated as auxiliary variables. $\mathbf{v}(t)$, $\omega(t)$ and $\mathbf{I}^{-\mathbf{1}}(t)$ are updated on every time step and are affected directly or indirectly by the simulation environment. Although, rotation matrix, $\mathbf{R}(t)$, presented in the equations is effectively replaced by quaternion $\hat{\omega}(t)$, $\mathbf{R}(t)$ is still utilized when inertia tensor is updated. Quaternions can be easily transformed to rotation matices and vice versa.

2.1.2 Time-advancement

State of each body advances in time steps of size dt. Time-advancement is performed by solving ordinary differential equation (ODE)

$$f(\mathbf{X},t) = \dot{\mathbf{X}}(t) = \frac{d\mathbf{X}(t)}{dt} \tag{2.5}$$

In practice, infinitesimal dt cannot be expressed computationally. Instead, some "small-enough" discrete time step Δt is used. Time advancement of state, $\mathbf{X}(t)$ from current time t_i to t_{i+1}, that is

$$\mathbf{X}_i \overset{\text{step}}{\Longrightarrow} \mathbf{X}_{i+1} \tag{2.6}$$

is equivalent to:

$$\mathbf{X}(t_0) \overset{\text{step}}{\Longrightarrow} \mathbf{X}(t_0 + \Delta t) \tag{2.7}$$

is also referred as time-stepping. Solving ODE numerically, and can be generally expressed as

$$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta\mathbf{X}, \tag{2.8}$$

and different ODE solvers approximate $\Delta\mathbf{X}$ differently, and each one of them have their advantages and disadvantages.

2.2 Numerical Integrators

In this section the numerical integrators used in the simulation are described. These integrators are approximations of integration of continuous systems used in solving ODEs related to real-life problems. This approximation holds as long as the time interval is small enough and the solver is convergent. All of the following solvers are implemented in simulation software, used in this research, and tested at some point, but only one is used in the actual measurements.

2.2.1 Euler's Mehtod

Euler's method, or explicit Euler's method, is a simple method for solving ODEs. It is expressed as:

$$\mathbf{X}_{i+1} = \mathbf{X}_i + \Delta t \dot{\mathbf{X}}, \tag{2.9}$$

where $\mathbf{X}$ is the state of the system and $\dot{\mathbf{X}}$ it's time derivative. Explicit Euler's method is considered inaccurate and it never converges ($\Delta t > 0$). Instead, it's divergence rate is directly proportional to the time-step size. Euler's method is not symplectic integrator and thus, it does not conserve energy in physics simulations.

2.2.2 Semi-Implicit Euler

Semi-Implicit Euler's (SIE) method, is an improvement of Euler's method. It is expressed as:

$$\begin{aligned} \mathbf{V}_{i+1} &= \mathbf{V}_i + \Delta t \dot{\mathbf{V}}_i \\ \mathbf{X}_{i+1} &= \mathbf{X}_i + \Delta t \mathbf{V}_{i+1}, \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} \mathbf{V}_i &= \dot{\mathbf{X}}_i \\ \dot{\mathbf{V}}_i &= \ddot{\mathbf{X}}_i \end{aligned} \tag{2.11}$$

SIE accumulates similar error to Euler's method, and is thus not accurate, but does not diverge, which is a clear advantage over its predecessor.

2.2.3 Runge-Kutta

Fourth order Runge-Kutta (RK4), reduces the error of Euler's method from $O(\Delta t)$ to $O(\Delta t^5)$.

$$
\begin{aligned}
k_1 &= \Delta t f(\mathbf{X}_i, t_0) \\
k_2 &= \Delta t f(\mathbf{X}_i + \frac{k_1}{2}, t_0 + \frac{\Delta t}{2}) \\
k_3 &= \Delta t f(\mathbf{X}_i + \frac{k_2}{2}, t_0 + \frac{\Delta t}{2}) \\
k_4 &= \Delta t f(\mathbf{X}_i + k_3, t_0 + \Delta t) \\
\mathbf{X}_{i+1} &= \mathbf{X}_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4.
\end{aligned} \tag{2.12}
$$

RK4 is accurate and often used, but does not conserve energy. Although, the method is very stable compared to Euler's method, and can thus be used for larger systems and with larger Δt, although it diverges slowly.

2.2.4 Verlet Velocity Integrator

Verlet Velocity Integrator (VVI) originally described by Verlet (1967) is less accurate than RK4, but stable. It produces $O(\Delta t^3)$ error and can be expressed as follows:

$$
\begin{aligned}
\mathbf{V}_{i+\frac{\Delta t}{2}} &= \mathbf{V}_i + \frac{\Delta t}{2}\dot{\mathbf{V}}_i \\
\mathbf{X}_{i+1} &= \mathbf{X}_i + \Delta t \mathbf{V}_{i+\frac{\Delta t}{2}}
\end{aligned} \tag{2.13}
$$

3 Bird's eye view on collision detection

The primary purpose of a collision detection engine is to detect if collisions between rigid bodies occur in the first place, and if they do, calculate contact points, normals and distances between colliding objects.

The collision detection process can be divided into two phases, the broad-phase and the narrow-phase. The former is responsible of querying the world for possible collisions and, if such events might be occurring, the latter handles the more detailed inspection of collisions between specific colliding object pair candidates.

3.1 Broad-phase

Purpose of the broad-phase is to assess if any pair-wise collision detection should be tested in the first place. Although, implementing an extra layer of algorithms increases the complexity of the collision engine, the benefit of the broad-phase is that it reduces the number of unnecessary collision queries, and the complexity of the querying algorithm from $O(n^k)$ to $O(\log n)$, where k is the dimensionality of the coordinate system, or $O(n \log n)$, depending on the underlying algorithm (Ericson 2004).

Typically, the game world is divided into spatial compartments. The objects residing in the same compartment, are queried for pair-wise collisions. If the objects inside the compartment are in close enough proximity, or close enough to the compartment walls, dividing non-empty compartments, the pair-wise collisions may be queried.

When the distance of the object and the illusionary compartment wall are calculated, a bounding volume (i.e. sphere) may be utilized to see if another object is near enough.

3.1.1 Octree

Meagher (1980) described octree as a data structure for presenting and querying three dimensional objects, and it is often used for acceleration structure for ray-tracing (Revelles, Ureña, and Lastra 2000), rasterization (Laine and Karras 2010) or some other static rendering process. In short, octree is a data structure that represents hierarchically evenly divided three dimensional space.

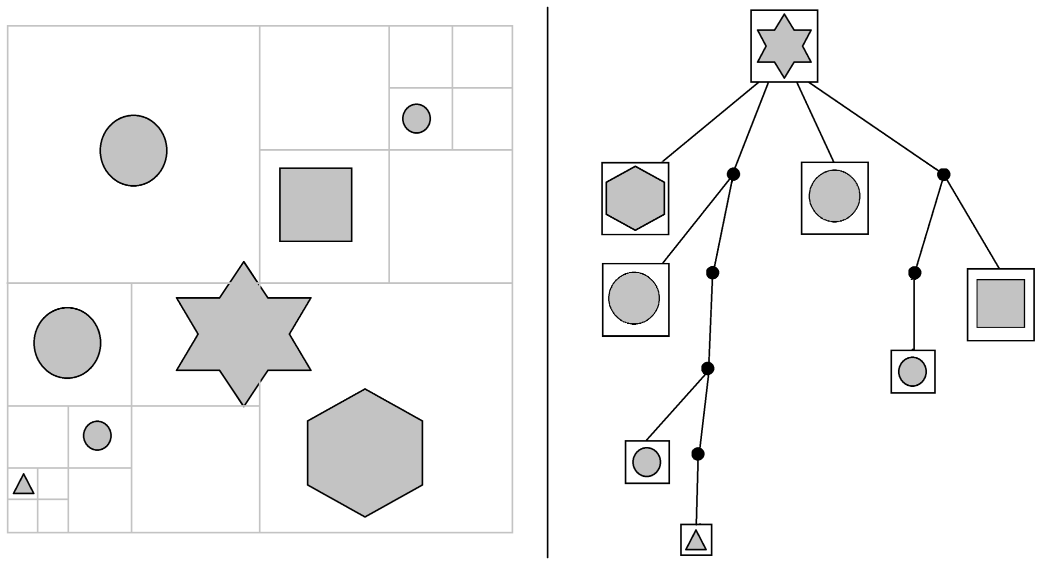

Figure 3. Principle of quadtree demonstrated in two dimensions. On left, the spatial partition is performed and on right, that partition is represented as a data structure. Note that on the left side, the space is divided as many times as needed in order to find the smallest space that still contains the child shape. The hexagonal star shape is stored in root since it does not fit to any child container.

Generally, an octree can be constructed and maintained based on different criteria. Their implementation details can be governed by the use-case. In Figure 3, the principle of octree is presented as a two dimensional case. Depending on the use-case, stored objects can either be stored to the signle smallest container they fit or several, if they overlap multiple containers. The division of a container to smaller containers is often limited explicitly, by assigning a minimum sub-space size or maximum depth.

For systems large enough, the increased complexity produced by a broad phase collision detection will be overcome by the improved overall performance, as broad-phase reduces the complexity of collision queries from $O(n^3)$ to $O(\log n)$.

3.1.2 Other broad-phase methods

BVH - Bounding volume hierarchy (Gu et al. 2013). In BVH, the game world is sectioned in volumes residing within other volumes, forming a tree structure similarly to octrees, with the exception that the sub-space containers are not necessarily cubic in shape, restricted to fill the parent container or equal in volume to their siblings. Bounding volume hierarchies have been studied as acceleraton structures in collision detection by Xiao-rong, Meng, and Chun-gui (2009) with AABBs, by Hubbard (1996) with spheres and by Gottschalk, Lin, and Manocha (1996) with OBB. The advantage of BVH over octree is flexibility in depth management. Also, in BVH, the containers can be moved during the simulation if possible. However, additional flexibility also adds complexity to updating the tree.

k-d tree. Originally developed by Bentley (1975), k-d tree has been used in collision detection, as was shown by Schauer and Nuchter (2015), and also as an optimization data structure for ray-tracing, which was demonstrated by Brown (2015). K-d tree, is a binary tree, in which the space is partitioned in half-spaces by planes. The planes are residing in branch nodes and the actual formed AABBs in leaf nodes. The partitioning plane and its position on the perpendicular axis is governed by heuristics, and the space division of a node is always performed perpendicular to its parent division plane. Unlike in BVH, the child containers always fill the parent fully, that is, the sum of volumes of all child half-spaces equals the volume of the entire space.

Sweep and prune. Possibly the most common broad-phase collision detection algorithm was originally formalized by Baraff (1992) in his PhD thesis. In sweep and prune, the extremities of objects are projected to axes, determined by the face normals of involved objects. For each object, for each projection axis there exists a lower bound and upper bound, and these values are sorted in a list, and queried for overlap. If a lower boundary of one object is higher on the list than a higher boundary of another object, these objects intersect with respect to this particular axis. In principle, on each frame, all the axis orientations and projections should be re-calculated and the list containing the boundaries, sorted or re-implemented completely. However, this computationally possibly cumbersome process is almost always avoided by taking advantage of temporal coherence. Originally, the algorithm was referred as *sort and sweep*, but in the process of publishing their physics engine I-COLLIDE, Cohen et al. (1995) rephrased the algorithm. More recently, (Karras 2012) also implemented sweep and prune to CUDA in order to utilize GPUs for collision detection.

3.2 Narrow-phase

As told in the previous section, the broad-phase queries the object pair candidates for possible collisions. Then, the narrow-phase uses the output of the broad-phase as input, and acquires information about the possible collisions, in more detail. The minimum requirement of a narrow-phase method is to detect if a contact between two objects occurs, but it may also provide some preliminary information for contact manifold construction, such as contact point, contact distance, penetration distance and contact normal. Sometimes, even the type of contact features, that is vertices, edges and faces, participating on the observed contact can be deduced.

4 Gilbert-Keerthi-Johnson algorithm

Gilbert, Johnson, and Keerthi (1988) came up with a procedure, for querying pair-wise collisions of polytopes by taking advantage of Minkowski difference of convex objects A and B:

$$\begin{aligned} A - B &= C \\ C &= \{\mathbf{a}_i - \mathbf{b}_j = \mathbf{c}_k\} \end{aligned} \tag{4.1}$$

where

$$\begin{aligned} i \in I \wedge |I| &= |A|, \\ j \in J \wedge |J| &= |B|, \\ k \in K \wedge |K| &= |I||J| \text{ and} \\ \mathbf{a}_i \in A, \mathbf{b}_j &\in B, \mathbf{c}_k \in C \end{aligned} \tag{4.2}$$

That is, the shapes of polytopes A and B are defined by a set of vertices **a** and **b**, respectively. The Minkowski sum, C, of sets A and B contains $|A||B|$ vertices. If A and B are convex, also the hull of C is convex. If origin is contained by the hull of C, that is $Conv(C)$, polytopes A and B are intersecting. The simplest way to demonstrate this property is to choose A and B to be single vertex polytopes, that is, points.

$$\begin{aligned} A = \{a_0\} \wedge B &= \{b_0\} \\ => A - B = C &= \{c_0\} \end{aligned} \tag{4.3}$$

$$c_0 = a_0 - b_0 = \begin{cases} \text{contact} & (0,0,0) \\ \text{no contact} & \neq (0,0,0) \end{cases} \tag{4.4}$$

The only way there can be a contact between A and B is if they populate the same point in space. This holds for more complex shapes also. However, in for more convex shapes the origin may or may not reside on the hull but within the hull, instead. Calculating the entire Minkowski difference of a pair of polytopes each frame, has a negative impact on performance. Fortunately, calculating the whole difference is not necessarily. In three dimensions, if any four points are selected from C a tetrahedron is formed that is fully contained within $Conv(C)$. Therefore, it is enough to find a simplex within $Conv(C)$ that contains the origin.

Traditionally, GJK also measures the minimum distance between polytopes and provides the closest points in A and B, participating in a contact. An implementation of GJK main algorithm is shown in Algorithm 3.1.

Algorithm 4.1 Gilbert-Keerthi-Johnson Algorithm (Bergen 2003)

```
$k = 0$
$W_k = \emptyset, \tau_k = \emptyset, \mathbf{w}_k = \mathbf{v}_k =$ arbitrary vector
**while** $|W_k| \leq 4$ or $||\mathbf{v}_k||^2 \leq \varepsilon^2 max\{||\mathbf{y}_k||^2 \, : \, \mathbf{y}_k \in W_k\}$ **do**
    $k = k + 1$
    $\mathbf{w}_k = s_{A-B}(-\mathbf{v}_k)$
    **if** $||\mathbf{v}_k||^2 - \mathbf{v}_k \cdot \mathbf{w}_k \leq \varepsilon^2 ||\mathbf{v}_k||^2$ **then**
        **continue**
    **end if**
    $\tau_k = \{\mathbf{w}_k\} \cup W_{k-1}$
    $[W_k, \lambda] = doSimplex(\mathbf{w}_k, \tau_k)$
    $\mathbf{v}_{k+1} = \Sigma_i \lambda_i \mathbf{y}_i$
**end while**
**return**$||\mathbf{v}_k||$
```

In Algorithm 4.1, k stands for iteration count, $\mathbf{w}_k$ is a new simplex point for iteration k, $\mathbf{v}_k$ is a search direction for iteration k, τ_k is a group of vertices resembling the simplex, $W_k \subseteq \tau_k$ stands for simplex primitive, λ is the set of barycentric coordinates, each $\mathbf{y}_i$ stand for a simplex witness point, and s_{A-B} resembles the support function, which is described in more detailed below.

In short, the origin is searched by building a 3-simplex, τ_k, and testing if the simplex contains the origin. On each k, the search direction, $\mathbf{v}_k$, is updated to point toward the origin and a new vertex, $\mathbf{w}_k$, furthest in $\mathbf{v}$ contained in Minkowski difference of A and B, is added to the simplex. τ_k is then passed to *doSimplex*, which evaluates the closest features of the existing simplex to origin, and makes assessment if the simplex should be evolved towards higher dimensions or if a vertex should be removed. The overall workflow of GJK is depicted in Figure 4.

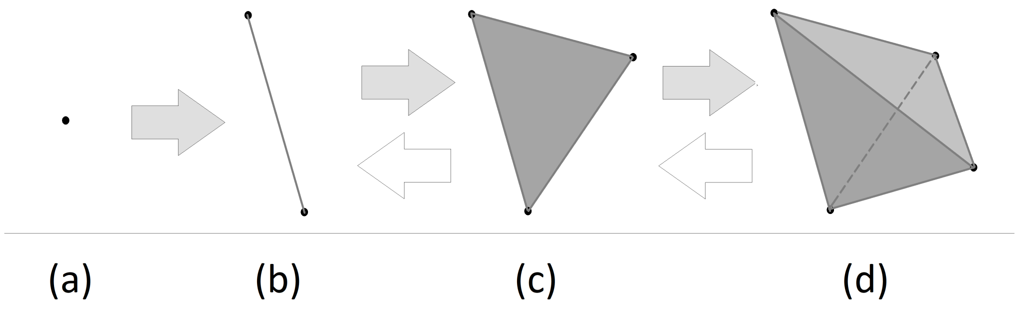

Figure 4. The overall flow of GJK algorithm. (a) 0-simplex. On the first iteration, usually another vertex is added automatically to the simplex, and the algorithm never comes back to (a). (b) 1-simplex. Depending on subroutine *doSimplex*, add another vertex or quit GJK. (c) 2-simplex. Depending on subroutine *doSimplex* remove a vertex and return to (b), add a vertex and proceed to (d) or quit. (d) 3-simplex. GJK finishes with success or backs down to (c) or (b) or quits.

In practice, $\mathbf{w}_k$ is found with the help of a support function, $s_{AB}(-\mathbf{v})$, which is defined as:

$$s_{A-B}(-\mathbf{v}) = s_A(\mathbf{v}) - s_B(-\mathbf{v}) \tag{4.5}$$

Here, $s_A(\mathbf{v})$ and $s_B(-\mathbf{v})$ are the support functions for searching the vertex contained in A furthest in direction $\mathbf{v}$, and the vertex contained in B furthest in direction $-\mathbf{v}$, respectively. The difference of return values of these functions equals the vertex furthest in direction $\mathbf{v}$ contained in $A - B$.

GJK can be applied to other applications such as ray-tracing (Bergen 2004) and proximity querying (Bergen 2001). Although, the core algorithm (Algorithm 4) for each application is the same, the *doSimplex* sub-algorithm is governed by the problem domain. Originally Gilbert, Johnson, and Keerthi (1988) used Johnson's distance sub-algorithm for querying contact events.

4.1 Johnson Distance Sub-Algorithm

In Johnson algorithm solves, in each iteration, a system of linear equations for each $W_k \subset \tau$, formed by simplex vertices $\mathbf{s}_i \in \tau_k$, for and barycentric coordinates λ, in a following way:

$$\begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ (\mathbf{s}_j - \mathbf{s}_l) \cdot \mathbf{s}_1 & \dots & (\mathbf{s}_j - \mathbf{s}_l) \cdot \mathbf{s}_r \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_r \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix} \tag{4.6}$$

or equally:

$$\mathbf{A}\lambda = \mathbf{b} \tag{4.7}$$

And barycentric coordinates are used for calculating the closest points $\mathbf{x}_i$ between objects,

$$\mathbf{v} = \Sigma_i \lambda_i \mathbf{x}_i \forall i = 1, ..., r \tag{4.8}$$

For barycentric coordinates, the following rule applies:

$$\Sigma_i \lambda_i = 1 \wedge \lambda_i \geq 0 \forall i = 1, ..., r \tag{4.9}$$

In the equations above, $r = m + 1$, where m is the simplex cardinality. That is, $m = |\tau_k|$.

Eq 3.7. ensures that for $W \subseteq \tau_k$, meaning that W represents a primitive of the simplex, such as point, line or plane, so that m for $W \leq m$ for τ_k, $\nu(\text{aff}(W)) = \nu(conv(\tau_k))$. The key idea in Johnson's algorithms is to search for point $\nu(\tau_k)$, or the point of minimum norm, which governs the search direction, $\mathbf{v}_k$. Geometrically, Johnson's algorithm deals with Voronoi regions of simplices. In each iteration new $\nu(\tau_k)$ is found, and it is always the point closest to the origin. In which primitive $\nu(\tau_k)$ can be found, is determined which primitive is closest to origin. This is continued until one of the exit conditions in Algorithm 4 is reached.

The overall flow of GJK algorithm could be described from the point of view of Johnson's algorithm as follows: First, the simplex is composed of only one vertex, $\tau_k = \{\mathbf{s}_1\}$. At this stage, $\mathbf{s}_1$ is the point of miminum norm, and the direction is set to $\mathbf{v}_k = -\mathbf{s}_1$. $\mathbf{v}_k$ is passed to support function $s_{AB}(-\mathbf{v}_k)$ and a new point, found furthest in $\mathbf{v}_k$ in the Minkowski difference, is added to the simplex yielding $\tau_k = \{\mathbf{s}_1, \mathbf{s}_2\}$. Next, Johnson's sub-algorithm performs a test, which primitive of the simplex, that is the $\mathbf{s}_1$ or $\mathbf{s}_2$ or the line formed in between them,

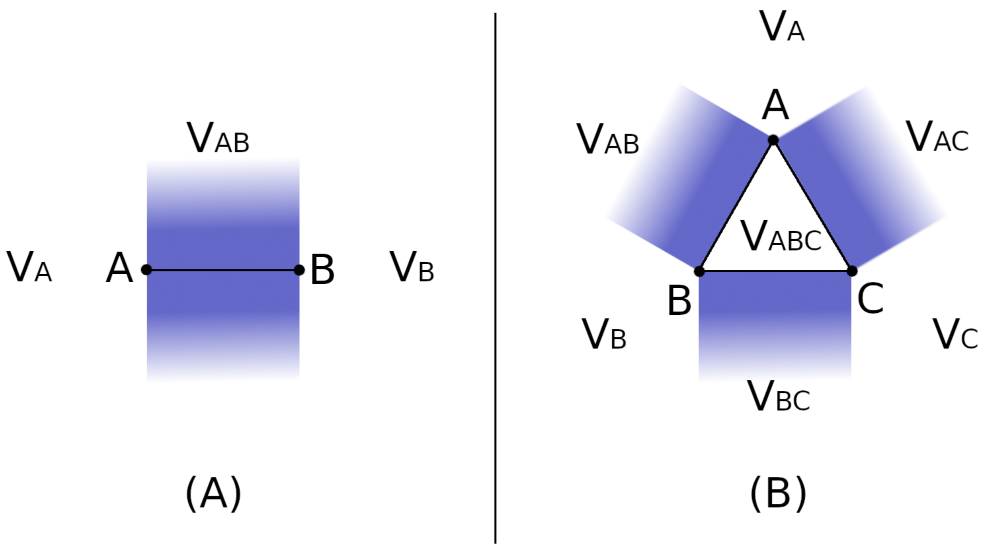

Figure 5. Voronoi regions of (A) line, or 1-simplex and (B) triangle, or 2-simplex. V_i, V_{ij} and V_{ijk} represent Voronoi regions of vertex i, line ij and triangle ijk, respectively, and i, j and k, the vertices corresponding to the simplex region. Although Voronoi regions of tetrahedron are also crucial to Johnson algorithm, they have been left out firstly, because their graphical representation can be complicated, and secondly because they are merely combinations of (A) and (B).

is closest to the origin, and calculates the $\nu(\text{aff}(W))$. This is depicted in Figure 5A, where V_A and V_B stand for Voronoi regions of end vertices and V_{AB} the Voronoi region of the line. The region inhabited by origin, dictates the closest primitive. If the origin is found closest to $\mathbf{s}_2$ of the end points, the other end point is replaced with a new point, found in direction $-\mathbf{s}_2$. If the origin is closest to the line, $\nu(\text{aff}(W))$ is calculated, and a new point, $\mathbf{s}_3$ is added to the simplex. Currently, the simplex is a triangle. Johnson's sub-algorithm performs a test for Voronoi regions of the triangle that has not yet been tested. If origin resides in the vicinity of $\mathbf{s}_3$, or in V_C in Figure 5B, one of the previous points $\mathbf{s}_1$ or $\mathbf{s}_2$ is removed and the previous phase for line testing is performed again. If origin resides outside the triangle, but in the close vicinity of yet untested edges, the point that is not part of the edge is removed and another point in the direction perpendicular to the edge closest to the origin is added to the simplex. Finally, if the origin resides within the triangle, or within V_{ABC}, Johnson's algorithm tests which side of the triangle origin resides, sets triangle normal as $\mathbf{v}_k$ and adds fourth point to τ_k. In the last phase, Johnson's algorithm tests if the simplex, which is now a tetrahedron, contains the origin. If the origin resides outside of the simplex it can reside in one of the seven different Voronoi regions, three edges, three faces and a vertex. If origin is contained in the simplex, objects A and B intersect. In short, Johnson algorithm performs a bottom-up query of Voronoi regions of a simplex, and as a result finds the point of the minimum norm and barycentric coordinates.

In practice, the required plane tests can be performed as dot products and the simplex and its primitives handled as concrete geometrical objects, but originally the algorithm was designed to calculate determinants and their cofactors, for plane tests, but also for the calculation of barycentric coordinates. However, both approaches, geometrically intuitive and determinant - based approach, are effectively equivalent.

In addition to Johnson algorithm, GJK constitutes a backup procedure to terminate the algorithm prematurely if a degenerate simplex is constructed. In degenerate cases, j:th cofactor of $det(W)$, $\Delta_j det(W)$ may be close to zero, and thus τ_k affinely dependent. The backup procedure determines $\nu(conv(W))$ by defining $\nu(\mathrm{aff}(W_s))$ for all $W_s \subset W$. Backup procedure always succeeds, but it is computationally more cumbersome than Johnson algorithm (Gilbert, Johnson, and Keerthi 1988).

Also, Johnson's distance algorithm fails due to numerical error when $|det(\mathbf{A})|$ becomes very small. Montanari, Petrinic, and Barbieri (2017) experimented on these issues and noticed that these issues arise from the implicit orthogonality requirement in Johnson's sub-algorithm (Eq 3.5). This is further enforced by requirements presented in Eq 3.7.

4.2 Evolution of GJK

GJK has been under extensive investigation and improvement on several occasions in the past. Ong and Gilbert (1997) used information of the adjacent vertices and Lin-Canny algorithm(Lin 1993) to improve the performance of the GJK. They updated GJK to take advantage of the temporal coherence, that is, it assumes that most of the time the system is not changed significantly between two consecutive frames and the contact information obtained in the last frame is still valid. The performance was noted as "excellent" (Ong and Gilbert

1997) when the object motions remained coherent, but a decrease in performance was also noted when coherence was lost.

Bergen (1999) developed an improvement to GJK coined as *Incremental Separating Axis* - GJK (ISA-GJK). ISA-GJK improves the original GJK by taking advantage of a few numerical enhancements. First, in ISA-GJK, the backup procedure is completely removed and is replaced by caching support points, simplices, dot products and determinants from last iteration. Bergen (1999) state that when caching is done properly the algorithm may terminate within the first iteration. In terms of the simulation, this procedure exploits temporal coherence. Second, in order to know if a collision is occurring between a pair of objects, it is enough to know if $\mathbf{v}_k \cdot \mathbf{w}_k > 0$. Calculating vector length involves taking square root which is an expensive operation. By utilizing $\mathbf{v}_k \cdot \mathbf{w}_k > 0$, or squared lengths (Algorithm 4.1), the performance of GJK can be further improved. With these principles the core GJK algorithm can be reduced to Algorithm 4.2.

Algorithm 4.2 ISA - GJK Algorithm (Bergen 1999)

```
$k = 0$
$W = \emptyset$
while $\mathbf{v}_k = 0$ do
    $k = k + 1$
    $\mathbf{w}_k = s_{A-B}(-\mathbf{v}_k)$
    if $\mathbf{v}_k \cdot \mathbf{w}_k > 0$ then
        return false
    end if
    $\mathbf{v}_k = v(conv(W_k \cup w_k))$
end while
return true
```

Casey Muratori showed in *Implementing GJK* (2006) how to apply GJK for collision queries, when the only interest is whether a collision occurs or not. In this approach, GJK is stripped down of all unnecessary functionality. Practically, this means neither barycentric coordinates, nor distance information is extracted. This implementation has also referred as *boolean GJK*(Linahan 2015). Similarly to ISA-GJK, in boolean GJK the termination condition is $\mathbf{v}_k \cdot \mathbf{w}_k > 0$. Also, in *Implementing GJK* (2006), the GJK is portrayed in an intuitive geometric manner. This implementation was later published by Linahan (2015).

If boolean GJK is intended to be used in a collision detection engine, an additional the helper algorithm, expanding polytope algorithm (EPA)(Bergen 2003) or a method applying separating axis theorem (SAT)(Boyd and Vandenberghe 2004), is needed to obtain contact information and construct manifolds.

4.3 Signed-Volume Sub-Algorithm

Recently, Montanari, Petrinic, and Barbieri (2017) came up with a sub-algorithm, Signed-Volume, that would overcome the numerical downsides of Johnson algorithm in a performance increasing manner. They found a way to preserve the orthogonality condition embedded in Eq. 4.6. by moving it from the left side to the right side of the equation, yielded an algebraic system:

$$\begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ s_1^l & \dots & s_{r+1}^l \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_r \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ p^l \end{bmatrix} \tag{4.10}$$

or equally:

$$\mathbf{M}\lambda = \mathbf{p} \tag{4.11}$$

In Eq. 4.10 **M** is a matrix formed from simplex τ. The equation states that by projecting a simplex M on a plane, and a triangle is formed, and that triangle contains the point **p**. This projection can be done because the barycentric coordinates are invariant under projection, which allows the calculation of barycentric coordinates in lower dimensional space. The algorithm has three main phases:

1. Vertices of simplex, $\tau \in \mathbb{R}^m$, are projected into $\mathbb{R}^r$, where $r \leq m$.
2. Discard any vertices not supporting $\nu(\tau)$.
3. Solve Eq. 4.10 for λ.

Signed-Volume prevents the multiplication of potentially small quantities, and thus most of numerical issues exhibited by Johnson algorithm.

As described in previous subsection, in Johnson algorithm, the algebraic system Eq. 4.6 is solved for each $W \subset \tau$. However, in Signed-Volume, a unique set of points for which the Eq. 4.10 is solved for barycentric cordinates. Both algorithms must satisfy Eq. 4.9.

5 Bird's eye view on collision detection revisited

The latter chapter related to the general flow of collision detection process briefly introduces some additional, but also generally used, methods. Mostly, it focuses on how the information obtained from the actual detection query is used further in the process.

5.1 Contact Handling

5.1.1 Manifold Construction

After a collision between two objects is detected, and contact distance, or penetration depth, and the closest points have been found by narrow-phase, a contact manifold can be constructed. The contact primitive, that is, vertex, edge or face, participating on a contact is solved at first, for both objects. This can be achieved by exploiting the number of different support points per object needed to construct the final simplex. In GJK, support points for both objects are stored into a list. The size of the list of distinguish support points per object can be one, two or three, or four. These correspond to vertex, edge and face contact of that particular object, respectively. For example, when a collision between objects A and B is detected, if the number of different support points for object A is one and the corresponding number for object B is three the collision occurs between a face in object A, closest to object B, and a vertex in object B, closest to object A.

After the contact primitive types of both incident bodies are resolved the overall contact surface is clipped. This clipped surface holds the information about the contact points participating on the contact between the two bodies. This area is also known as the contact manifold, and information contained in the manifolds is used as an input for the collision response calculations.

5.1.2 Collision Response

The multi-point-contact method used in this research was originally presented by Giang, Bradshaw, and Sullivan (2003). The idea, in overall, is to solve:

$$\mathbf{b} + \mathbf{A}\mathbf{x} \geq 0 \tag{5.1}$$

for $\mathbf{x}$.

In the above equation $\mathbf{A}$ and $\mathbf{b}$ are term with a set of constraints and normals between a pair of contacts and between contact pairs, respectively. $\mathbf{x}$ is the set of contact forces to be applied to the bodies involved in incident contact. More detailed, Eq. 5.1 can be expressed as:

$$b_i = \Sigma_j x_j A_{ij}, \tag{5.2}$$

where i is the contact index. Here, b_i is derived from distance constraint requirements.

$$\begin{gathered} \mathbf{c} = \mathbf{p}_B - \mathbf{p}_A = 0 \\ \mathbf{p}_k = \mathbf{x}_k + \mathbf{R}_k \mathbf{r}_k \end{gathered} \tag{5.3}$$

and the time derivative:

$$\begin{gathered} \dot{\mathbf{c}} = \mathbf{n}_i(\mathbf{v}_B - \mathbf{v}_A) \geq 0 \\ \mathbf{v}_k = \dot{\mathbf{p}}_k = \mathbf{v}_k + \omega_k \times \mathbf{r}_k \end{gathered} \tag{5.4}$$

being:

$$\begin{gathered} b_i = v_{rel,i}^{-}(1 + \varepsilon) \\ v_{rel,i}^{-} = \mathbf{n}_i(\mathbf{v}_B^{-} - \mathbf{v}_A^{-}) \end{gathered} \tag{5.5}$$

Where $v_{rel,i}^{-}$ is the relative velocity between two bodies about the contact normal n_i for i:th contact. A minus sign (-) and a plus sign (+), in the superscript, signifys a before collision value and after collision value, respectively.

The elements of matrix $\mathbf{A}$ stems from equations of motion and, in short, can be described as:

$$A_{ij} = \mathbf{n}_i \cdot \left(\left(\frac{\mathbf{n}_j}{M_{A_i}} + \mathbf{I}_{A_i}^{-1}(\mathbf{r}_{A_i} \times \mathbf{n}_j) \times \mathbf{r}_{A_j}\right) - \left(\frac{\mathbf{n}_j}{M_{B_i}} + \mathbf{I}_{B_i}^{-1}(\mathbf{r}_{B_i} \times \mathbf{n}_j) \times \mathbf{r}_{B_j}\right)\right) \tag{5.6}$$

It should be noted that in order to get proper values for contact forces or impulses in the simulation. All the contacts must be solved simultaneously with a linear equation solver or, for inequality condition, with an LCP-solver, which is described in more detail in the next section.

5.2 Solvers for Systems of Equations

After contacts are resolved and contact manifolds constructed, the contact response is handled and, usually, a system consisting of multiple variables is evaluated. Here, some most common methods and tools, which are also used in the context of this thesis, are presented.

5.2.1 Linear equations

$$\begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix} = \begin{pmatrix} a_{11} & \dots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \dots & a_{kk} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} \tag{5.7}$$

Which can be written as:

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{5.8}$$

And to solve all the x_i in $\mathbf{x}$:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{5.9}$$

In the context of physics simulations, each x_i signifies an impulse or a force for *i*:th contact. Even though, it would be simple to write a brute-force -solver for eq 2.13., pre-existing software libraries, such as LAPACK (*LAPACK documentation.* 1992) or Eigen (*Eigen documentation.* 2009), of which the latter is utilized in the implementation part of this thesis, contain highly optimized tools for solving systems presented here, and they are freely available. The corresponding tool set for an NVIDIA chip set is found, for example, in cuSOLVER library (*cuSOLVER documentation.* 2007).

5.2.2 Quadratic

In physics simulations, quadratic solvers are used for systems described with quadratic behavior that are subjected to linear constraints. Quadratic optimization problems are generally expressed in terms of optimized function and constraints.

$$\begin{gathered}\frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x}+\mathbf{c}^T\mathbf{x}\\ \mathbf{A}\mathbf{x}\geq\mathbf{b}\end{gathered}\tag{5.10}$$

Here the above function represents the behavior, and the lower the constraint, to which the behavior is subjected to. As described by Baraff and Witkin (1997), quadratic solvers can be used in contact resolution, for example, in a following manner:

$$\begin{gathered}\ddot{\mathbf{d}}=\mathbf{A}\mathbf{f}+\mathbf{b}\\ \mathbf{f}\cdot\ddot{\mathbf{d}}=0\end{gathered}\tag{5.11}$$

In the above equation $\ddot{\mathbf{d}}$ and $\mathbf{f}$ are vectors containing the relative acceleration between objects of a contact pair, and the corresponding applied forces, respectively. The vectors with superscripts $\mathbf{x^T}$ and $\mathbf{c^T}$ are transposes of vectors $\mathbf{x}$ and $\mathbf{c}$, respectively. Matrix $\mathbf{A}$ and vector $\mathbf{b}$ are described above. The system simply describes that if contact remains between contact pairs of contact i, then $\ddot{d}_i=0$ and $f_i>0$, and if the contact breaks, $\ddot{d}_i>0$ and $f_i=0$.

Millington (2010) describes the entire structure and functionality of their physics engine without usage of quadratic solvers. Also, Baraff and Witkin (1997) state that contact handling in constrained physics simulations is usually handled as *linear complementary problem*.

5.2.3 LCP

As stated above, it is more common to solve contacts, in physics engines, as linear complemetary problems than general quadratic problems. In fact, linear complementary problems are a special case of quadratic problems, and LCP methods can be used for solving them. An example of common LCP solver algorithms are Projected Gauss-Seidel and Projected Jacobi. Both of these algorithms are described by Richards, Benevolenski, and Voroshilov (2012).

LCP can be generally expressed as:

$$\begin{aligned} \mathbf{Af}+\mathbf{b}, \mathbf{z} \geq 0 \\ \mathbf{z}^T \cdot (\mathbf{Af}+\mathbf{b}) = 0 \end{aligned} \tag{5.12}$$

Whereas quadratic programming attempts to find the smallest values for all elements of **f**, LCP solvers attempt to find **f** that solves both of the constraints.

5.3 Other narrow-phase methods

5.3.1 Separating axis theorem

The separating axis theorem (SAT) is a narrow-phase technique based on hyperplane separation theorem (Boyd and Vandenberghe 2004). The idea is to project the extremities of both of the colliding objects of the object pair to a set of axes and see if the objects overlap with respect to that axis. These axes are defined by face normals (edge normals in 2D) of both of the objects. If the objects overlap with respect to all of the axes, they collide. Hornus (2017) compared a set of narrow-phase algorithms, and found that SAT performs worse than GJK, with a higher number of required plane tests per frame, that is for more complex polygons. Also, it has been mentioned by Ericson (2004) that SAT not provide any information about contact points or distance on its own. Therefore, it is understandable if GJK is more commonly used as a narrow-phase method by physics engines.

5.3.2 Minkowski portal refinement

Another convex polytope based method that uses support mapping is Minkowski Portal Refinement (MPR) algorithm invented by Snethen (2008), who has also stated that it is more robust, simpler and therefore more performant and maintainable than the original GJK, but it does not return the minimum distance between colliding shapes. MPR is implemented in Xenocollide library *Xenocollide library* (2006) by its author and it is freely available.

5.4 Approximate methods

As stated by Richards, Benevolenski, and Voroshilov (2012), in games, physics is not always simulated exactly. Dealing with forces and acceleration involve dealing with second order derivatives, which has an impact on both, the numerical stability and performance of the system as discussed by Bender, Müller, and Macklin (2017). A solution to this is solving only first order derivatives and approximating Newton's second law with momentum and velocity (Richards, Benevolenski, and Voroshilov 2012). The momentum-velocity model is more about dealing with the actual rigid body simulation and not just collisions and contact handling. Alternative term for the momentum-velocity model is position-based dynamics (Bender, Müller, and Macklin 2017).

The method is implemented in Box2D (Catto 2007), an open-source physics engine, by Erin Catto, along with the GJK. It was Catto, who also introduced sequential solver for solving contacts iteratively instead of solving linear equations or LCPs. Sequential solver is analogous to Projected-Gaussian Seidel, originally reported by Morales, Nocedal, and Smelyanskiy (2008), and its main advantage is that it makes contact handling simpler in terms of programming, as contact forces can be solved separately from the other simulation and simply added to the system.

Another advantage of the momentum-velocity model over traditional Newtonian-physics is that it provides an alternative method for resolving collision time. The way the collision time is resolved, with the methods described in above, it's usually heavily coupled to either to the rigid body simulation, collision detection or both. Firth (2011) introduced a method called speculative collisions. The idea is to take into account the movement of objects in the contact resolution step, by adding a small quantity of velocity to the relative velocity between objects. This is usually in the order of the time step of the simulation. This means, speculative contacts is a form of CA time handling method. Speculative time handling is implemented in an open source physics engine called JitterPhysics (*JitterPhysics - engine Readme*) and it can also be used in Unity3D (*Unity3D - Manual, Continuous Collision detection*).

6 Remarks on GPU Implementation

In this chapter, the hardware, principles and programming some conventions for a graphics processing unit (GPU) will be introduced, the background and its common usage is described. In addition, general purpose GPU applications, relevant to this thesis, are reviewed and GPU is compared to CPU.

6.1 Background and GPGPU

Ever since the release of programmable GPUs, processing graphics has been made more available to developers. The development of shader languages, along with the GPU hardware, has ignited new approaches to process graphics in GPU. This includes some programmable steps, additional to the traditional graphics pipeline consisting of vertex and fragment shader, such as programmable geometry and tessellation shaders. Along with the increased freedom over the GPU pipeline, there are more stages that can be used for other graphics related computations, such as shadow calculations as shown by Billeter, Sintorn, and Assarsson (2010), or lighting as demonstrated by Boubekeur and Alexa (2008).

Hardware of the GPU has become more powerful and more control over the usage of GPU resources has been provided for programmers, as graphics pipeline has evolved. This combined with the parallel nature of GPU has resulted in other purposes for GPU, such as in scientific computation as described in general level by Luebke (2008), machine learning as shown by Garg et al. (2019) and game physics calculations, of which the most commonly used commercial GPU physics engine in games is *PhysX SDK documentation* (2001) by NVIDIA.

6.2 GPU Hardware Architecture

This chapter is mostly based on Pascal Whitepaper, published by NVIDIA (2016b) and a handbook written by Wilt (2013). In order to understand how GPU calculations are performed, a bird's eye perspective to GPU hardware architecture must be given. The description will follow NVIDIA Pascal micro-acrhitecture. It is chosen because of its similarities to any modern GPU microachitecture, and because it is also the governing architecture of the GPU hardware used in this investigation.

The central functional component for GPU for GPGPU are the *Streaming Multiprocessors* (SM). Each SM contains an instruction cache, instruction buffers, dispatch units, warp schedulers, registers, L1 cache, texture processors and memory and multiple cores. SM also contains *Warp scheduler*, with a dispatch unit, which is the GPU equivalent of CPU control unit. Warp scheduler organizes the instructions in a single unit of a set of threads that share the same instructions. This unit is called a *warp*. The SM picks up the instructions from the warp sequentially. The execution is handled by GPU *Core*, which is a floating point unit (FPU). The SM structure of an NVIDIA GP100 series GPU is presented as a block diagram in Figure 6.

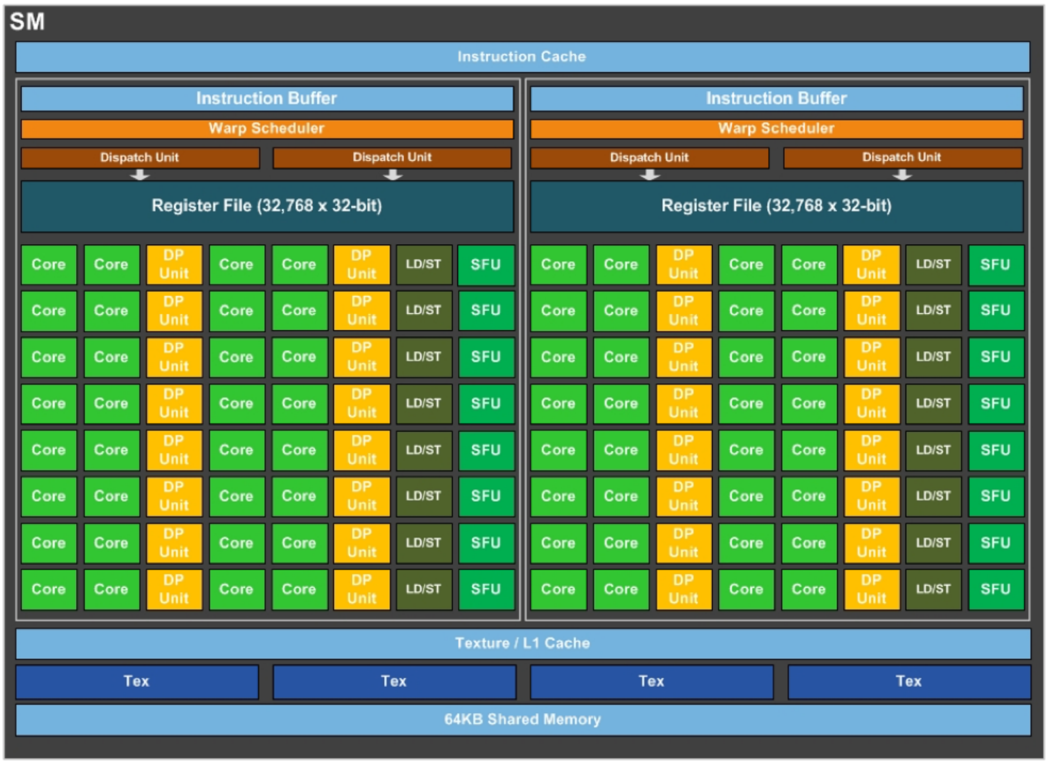

Figure 6. A block diagram of a single Streaming Multiprocessor (NVIDIA 2016b)).

32 bit single-precision cores, FPUs, in the Figure 6 are termed as CUDA. Each SM constitutes of 64 CUDA cores, and hardware model, GP106, used in our experiments, or commer-

cially GTX 1060, contains in total twenty SMs. In overall, this yields 1260 CUDA cores. A complete GP106 block diagram is presented in Figure 7. Here, the CUDA cores are presented in as green rectangles.

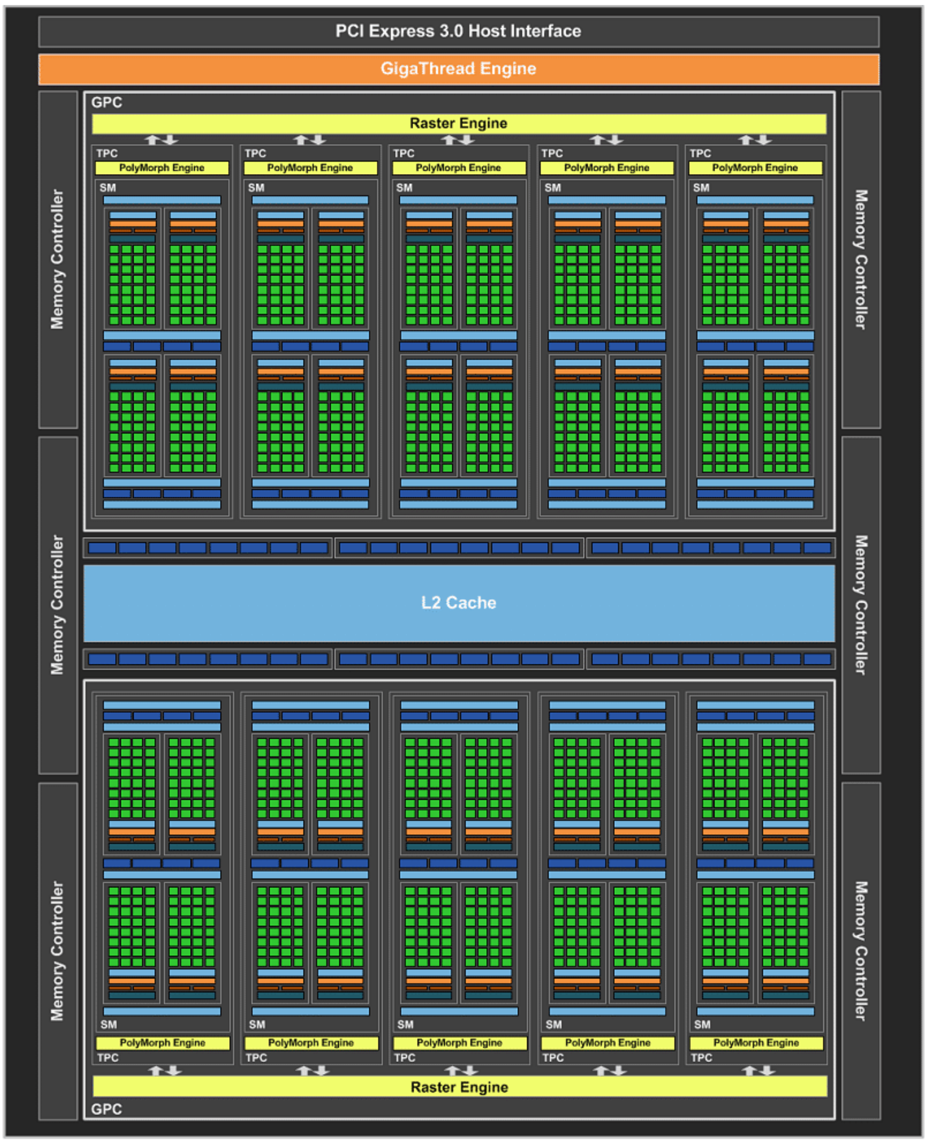

Figure 7. A complete GTX 1060 block diagram by NVIDIA (2016a)).

6.3 GPU Programming

The general execution pipeline in GPGPU calculations is independent of the API. In general, the execution is divided in sequential execution and parallel execution, performed in CPU and GPU, respectively. From programmer's perspective parallel execution is done in blocks and threads (Wilt 2013).

Cuda interface requires additional arguments when a function call is performed from CPU to GPU. These arguments are passed in a similar manner to C++ templates, but within triple brackets as in: `function<<<N, M>>>(args);`. In the example, `N` and `M` denote the number of blocks and *threads* per block, respectively, the block resembling a block of threads, or *threadblock*. That is, a programmer prepares the data and forms a *grid* of threadblocks, thus controlling the measure of parallelism used for the execution (NVIDIA 2019).

```
function<<<1, 1>>>(args);
```

would be completely sequential execution.

From hardware's perspective, a SM can execute multiple threadblocks and a thread is executed by a core. The grid represents the GPU unit used for the entire execution.

The threads and threadblocks are indexed, and commonly an execution of a function operating to an array of elements in parallel resembles the following:

```
__global__ void add(float *a , float *b , float *c , int N)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x ;
    if( index < N )
    {
        c[index] = a[index] + b[index] ;
    }
}
```

6.3.1 Code considerations

The rooted terminology in the field separates CPU and GPU executions as *host* and *device* executions. The data is prepared in host-side and possibly converted to a format that is excepted by the device. The memory allocations, on both, host and device, are done on host-side. The initialization of the device execution is performed by calling a kernel function. In CUDA, the *lingua franca* is C, or a subset of C, with some functionalities replaced with interfaces specific to the client-side, such as `malloc` with `cudaMalloc` and `memcpy` with `cudaMemcpy` (NVIDIA 2019). An example of a kernel function declaration and execution:

```
__global__ void my_kernel(int *a, int *b, int *c, int N);

int main()
{
        int *da, *db, *dc, N;
 // initilize data, do memory allocations
        int blocks, threads;
        my_kernel<<<blocks, threads>>>(da, db, dc, N);
        return 0;
}

__global__ void my_kernel(int *a, int *b, int *c, int N)
{
 // kernel function body
}
```

Also all function prefixed with a `__global__` keyword. The other prefixing keywords in CUDA are `__device__` and `__host__`. Kernel function can be thought of as a bridge between host and device execution. This is because device functions can only be called from other functions labeled with `__device__` or `__global__`, whereas `__global__` and `__host__` functions can be called from host (NVIDIA 2019).

In this research, the GPU-side GJK process is implemented using the following conventions:

```
// GJK for single contact pair
__device__ float gjkGPU(bdGPU *bodyA, bdGPU *bodyB,
        simplexGPU *simplex, float *distance);

__global__ void queryContacts(queryPairGPU *pair,
        float *distances, unsigned int Npairs)
{
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        int stride = blockDim.x * gridDim.x;
        for (int i = index; i < N; i += stride)
        {
```

```
// GJK for single contact pair
                distances[i] = gjkGPU(contactPairs[i].bodyA,
                contactPairs[i].bodyB, &contactPairs[i].simplex);
        }
}
...
void narrowPhase(queryPairGpu **contactPairArr,
        float **distances, unsigned int N)
{
        queryContacts<<<1,N>>>(*contactPairArr, *distances, N);
}

vector<Contacts> collisions(const QueryPairs &pairs)
{
        vector<queryPairGPU*> pairs;
        vector<float*> distances;
// build data, allocate memory

        queryContacts(pairs.data(), distances.data(), pairs.size());
// build and return contacts
}
```

6.3.2 Parallel thinking

The following terminology and principles are mainly based on the text written by Karras (2012), by using a BVH construction and traversal as an example. The factors pointed out in this chapter demonstrates, is how differently sequentially executed algorithms should be coded and their performance cost evaluated compared to parallel execution. Simple evaluation of algorithm complexity, in a form of *O(n)* or *O(log n)*, is not sufficient for evaluating GPU-run algorithms.

The first and foremost factor affecting GPU-execution that should be considered, both in computations and graphics applications, is *divergence*. Divergence is a measure of similarity of operations performed by close by threads, and it's evaluated both for data and execution, yielding *data divergence* and *execution divergence*, respectively. If nearby threads are executing the same code or execution branches, the execution divergence is said to be low. Similarly, if the data is read from, or written to, a same location in memory or registers, the data divergency is low. The lower the divergence, the better it is for the performance of the algorithm. Of course, this is same for all parallel executions, not just GPU. In CPU executions this is not so clearly evident as cache size per core is high compared to GPU and cache is more easily available by the cores executing instructions. This is also referred as global cache coherence, originally described by Ravishankar and Goodman (1983).

Another factor affecting parallel GPU-executions is *occupancy*, which is a ratio of threads involved in execution per available threads on the processor, that is occupied active warps per maximum available warps. A new set of parallel instructions, that is a single thread block, are dispatched from a single warp, at a time, for the execution. If the warp is not fully occupied, the SM is not operating in full capacity, and a full parallellism isn't achieved. Occupation is especially important for large data size, capable occupying a significant amount of warps in the whole GPU.

6.3.3 Unified Memory

Originally, the GPGPU programmers using CUDA had to allocate memory separately for host and device side, initialize data on host side, use `cudaMemcpy` to copy the data to a memory address pointed by a device pointer and run the execution. After the execution, they had to use `cudaMemcpy` to copy the data back to host, and free host and device pointers, after data usage. This was because the CPU and GPU memory was physically separated and shared address space didn't exist. On CUDA 6, the Unified Memory (UM) was introduced. UM creates a pool of managed memory which can be accessed by both, CPU and GPU. However, this model required that all managed memory utilized by the host, has to be synchronized with the memory used by the device, before any kernel calls. The allocation of memory and usage of variable on host side, in terms of CUDA 6, works as follows:

```
int doStuff(int size)
{
        int *data;
// allocate unified memory
        cudaMallocManaged(&data, size);
//... init / handle data
        int blocks, threads;
        doStuffGPU<<<blocks, threads>>>(data, N);
// synchronizes the data and returns back
// to sequential execution
        cudaDeviceSynchronize();
// ... use data here...
// ... before releasing unified memory
        cudaFree(data);
}
```

Pascal architecture came after CUDA 6, and brought in two improvements: support for large space addressing and memory page faulting mechanism. Former enabling the GPU access to CPU and GPU virtual spaces. The latter guaranteeing a global data coherency between CPU and GPU. The unified memory in CUDA 9 are used in a following manner:

```
int doStuff(int size)
{
        int *data;
// allocate unified memory
        data = (int*)malloc(size);

... init / handle / data

        int blocks, threads;
        doStuffGPU<<<blocks, threads>>>(data, N);

// synchronizes the data and returns back
// to sequential execution
        cudaDeviceSynchronize();

... use data

// free unified memory
        free(data);
}
```

Although, CUDA 9.x is the current practice, the support for the post-CUDA 6 features is dependent on the operating system, and currently are only provided for Linux systems. Our experiments are conducted using Windows OS and thus, CUDA 6 practices will be followed, in terms of memory management, at least. Additionally, the GPU hardware utilized in the measurement (NVIDIA GTX 1060), supports CUDA 6.1 provided and earlier features, and the CUDA source will be compiled using `-arch=sm_61` flag.

6.4 GPU vs CPU

The term general-purpose graphics processing unit (GPGPU) calculation is an umbrella term for any computational tasks, other than graphics, performed using GPU. Handling physics in games with GPU can be considered as a GPGPU task. Handling physics with GPU frees one of the most computationally cumbersome tasks, in games, from CPU to GPU. Another reason than reallocating computational effort to GPU is that GPU is implicitly parallel. As described by Karras (2012), collision detection is a parallelizable task, and therefore suitable to be solved with a GPU. Using GPU over CPU, to solve parallelizable problems, may seem overwhelmingly advantageous. However, as discussed by Pabst, Koch, and Straßer (2010) the PCI-e bus creates a bottleneck in inter GPU data transfer resulting in a relative reduction in performance, meaning that doubling the number of cores in the GPU will not halve the computation time. Additionally, it has been noted by Pan and Manocha (2012) that when using only one core or thread the CPU has been found generally more computationally effective. Despite of these findings, the shear number of GPU cores is so vast that, in collision detection, the performance advantage can be more than one order of magnitude higher than solving the same problem with CPU, as was noted by Pabst, Koch, and Straßer (2010). Also, it should be noted that as described by NVIDIA (2016b) the CPU-GPU data transfer is not necessarily performed via a PCI-e bus, but with NVIDIA Pascal architecture, a more recent NVLink is possibly utilized. Pan and Manocha (2012) noted that benchmark comparison between CPU and GPU is not a trivial matter, and it's affected by the algorithm, CPU and GPU hardware and the amount of parallelism used in both CPU and GPU, and the also by the test set. These factors are further discussed in section *Results and Discussion*.

Additionally, as noted in CUDA Technical Documentation (NVIDIA 2019) and Pascal Whitepaper NVIDIA (2016b), there is no global cache coherence in Pascal. Only within each SM, which have their own L1 cache. Also there is a shared memory between cores within SM and the L2 is shared between SMs. This is very different from current CPU that thrives on cache coherency. In addition, memory size and bandwidths differ between CPU and GPU. The L1 and L2, of a current GPU, are more than one fold smaller, whereas the bandwidth is 2 fold, compared to CPU.

7 Experimenting with SV-GJK

In this chapter, the simulation software and the experiments conducted using the software are described and the experimental environment is presented. Also, a depiction of how the results will be presented, is given.

7.1 Description of Simulation Software

The simulation software is a typical physics simulation software, combined with a game loop, not unlike described by Nystron (2014), who also describes some other design patterns that influenced the implementation. In *Game*-class, while-loop is executed as long as the simulation is halted. This loop (Figure 2) consists of three major functions: Game::input(), Game::update(float), Game::render(), all executed sequentially. The first one handles player input, which in this simulation is only available for some debug functionality and handling camera movement. The second function, Game::update(float), does most of the work; during its execution forces, velocities and positions are calculated, collisions queried, contacts solved and impact from the previous frame applied. The last function in the loop is solely for rendering purposes.

The overall class diagram of the software is provided in Figure 8. The *main()*-function uses an instance of game class, which initiates all needed objects in the before the loop is run. During update the positions and physics of all instances of *Object*-class are updated. Those objects that move use an object of *Body*-class, which represents a rigid body and handles all physics, and an instance of *Shape*-class, that deals with rendering and shape related information such as vertex positions. SV-GJK and the applied acceleration structure is implemented in update-phase. The collision queries are abstracted in *CollisionDetection*-class. The GJK algorithm is implemented in a way that the sub-algorithm can be changed if needed. The collision information is saved to an instance of *Contacts*-class, which subsequently constructs the contact manifold. A contact contains data about contact point, contact normal and pointers to corresponding the colliding object pair. The constructed contact information is then passed to *ResponseHandler*-class, which solves contact impulses and passes them to the rigid bodies of incident objects, for the next frame.

The selected techniques implemented in simulation software, described in the theory part of this thesis, are VVI solver and octree as broad-phase method.

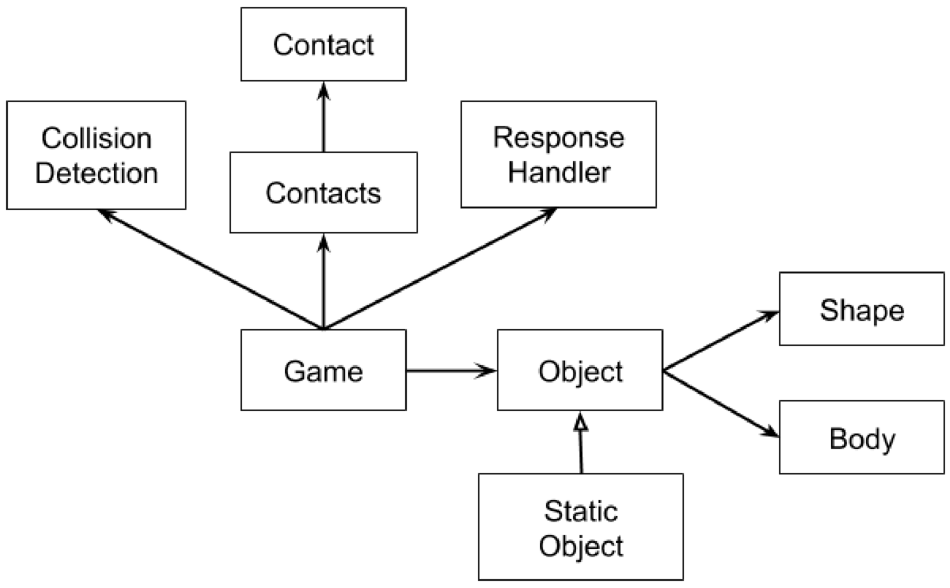

Figure 8. Class diagram of the simulation software. Only the most important classes are included in the description.

The renderer is OpenGL (v. 4.6), which is used with GLEW extension and GLSL shading language (version 460 core). The window handle is provided by GLFW extension, which also implicitly provides the needed game loop.

The original CPU implementation of the algorithm was done with C++. Since C++ is not supported by CUDA, this causes a possible incomparability between CPU and GPU executions. This issue was solved by using the implementation in *openGJK*-library by Montanari, Petrinic, and Barbieri (2017) for both GPU and CPU simulation. In addition, GPU profiling tools are not built for measuring performance of the GPU code executed as part of extensive MSVC projects. This problem was solved by using a custom *Logger*-class for both implementations.

7.2 CPU vs GPU Performance Comparison

GPU - CPU performance comparison is conducted using three different test cases. Each case consists of static object (i.e. "the floor"), which collides with other objects but does not have any other physics properties, such as velocity or momentum, and dynamic objects, which determine the nature of the particular measurement. Dynamic objects are perfect cubes. Such simple geometries are used in order to simplify the implementation and to reduce the number of complex collision schemes, which would make interpretation of the data more unambiguous.

In all experiments, the measured quantity is *contact query rate*, that is, observed contacts per time, as a function of the number of objects. Each measurement is set to run as long as a stable state in simulation is found. Another tested quality is parallelizability in GPU. Mainly, CPU will be run with single thread, and the optimal usage of cores and blocks on CUDA, in these test cases, is attempted to be uncovered.

The CPU execution time is measured by using C++ standard library tool std::chrono::high_precision_clock. With the utilized hardware, measurements can be realistically made with microsecond precision, which is sufficient. For GPU, the execution time is measured from host side (CPU) for the entire process, using the the same tool as for CPU measurements. Use-cases applied in the experiments were:

1. Static Tower - A tower of eight uniformly shaped cubes act as dynamic objects that are placed on a static object. Purpose is to test the performance of the algorithm for resting contact.
2. Dynamic Tower I - A tower, similar to Case 1, is assembled by dropping cubes from above. Purpose is to test the stability and robustness of the algorithm.
3. Dynamic Tower II - Same as Case 1, but an additional cube is tossed to the side of the tower to a random height from a randomized direction. The measurement is repeated numerous times in order to get enough data for statistical significance.

The CPU simulation is run with Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2501 Mhz, 4 Core(s), 4 Logical Processor(s) and the GPU code is executed by NVIDIA GeForce GTX 1060, 6 GB, with 32 CUDA FPU.

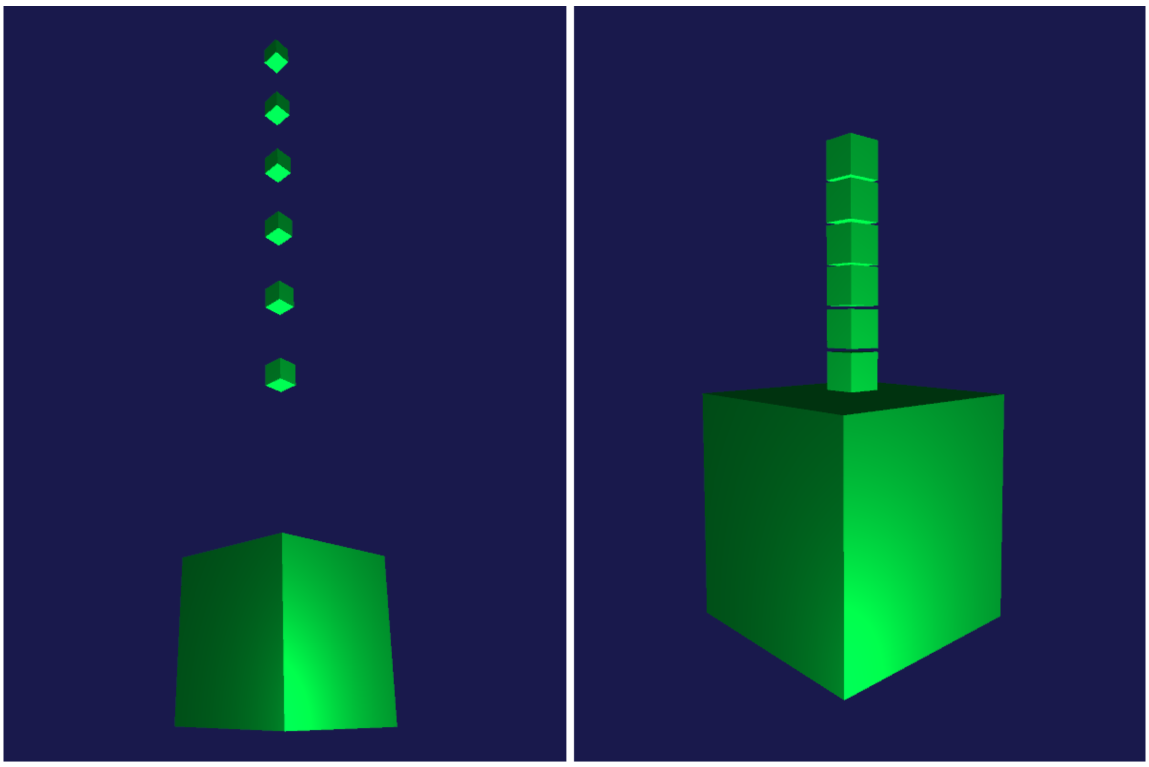

Figure 9. Screenshots of Case 2 performed with six dynamic cubes and one static (below). The initial state of the scene is presented on left, and the final state, when all cubes have fallen down and formed a pile, are shown on right. It should be noted that the camera can be moved similarly to first-person shooter games.

7.3 Results of Single-Core Experiments

The test cases 1 and 2 were conducted using up to 1 - 15 cubes. The results of a single CUDA core vs CPU measurement are depicted in Figure 10. The data of case 1 is plotted as an averaged contact query rate against sample size. Averaging was done over all frames. The same data was plotted in more general terms, as a ratio of collision detection rates of CPU and GPU (11). The data in acquired in case 1 is almost identical with the one obtained in test case 2. Only cases 1 and 2 were focused due to instabilities of Case 3, in which the outcome was not meaningful for any meaningful comparison.

Here, the results shown in Figures 10 and 11, conclude that in a single-core experiment, a CPU out performs the used GPU by being 2-5 times more performant. Also, the results show the decreasing efficiency in GPU side with an increasing object count. This performance

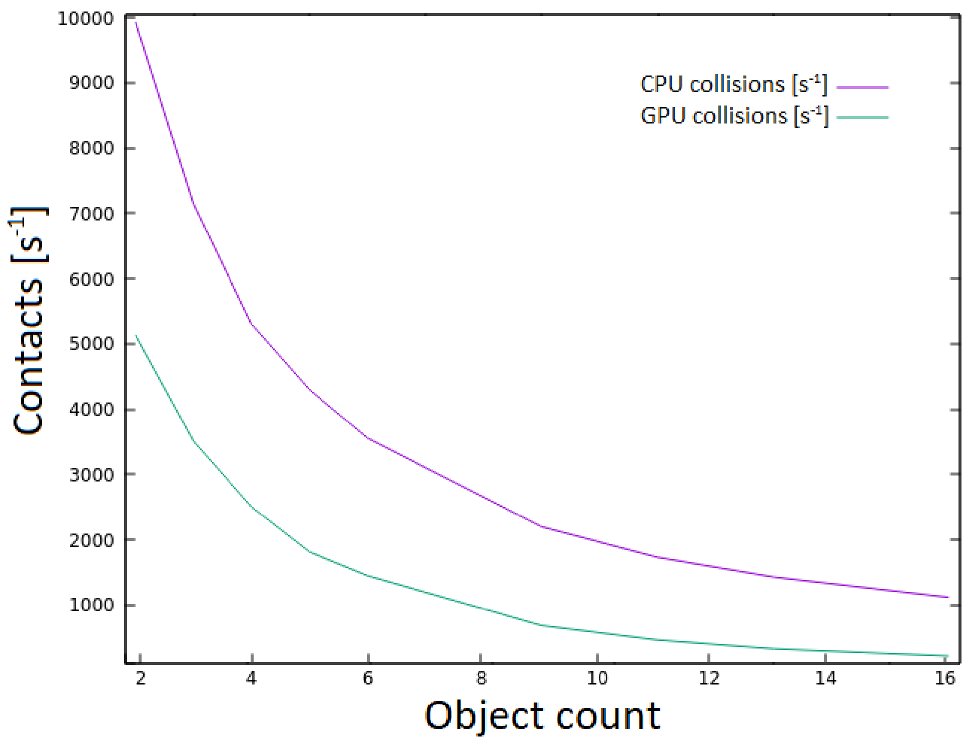

Figure 10. Comparison of a single-core performance. Average collision detection rate (Collisions / s) is depicted on the vertical axis and the number of cubes (8 vertices each) are represented by horizontal axis.

difference is largely explained by the technical differences between CUDA core and Intel i5 CPU, the latter being faster in terms of clock cycles (2.4 MHz vs. 1.5 MHz) and having 1-2 orders of magnitude larger cache L1 and L2 (NVIDIA 2016b). Pabst, Koch, and Straßer (2010) indicated that the performance could be hindered by the step transferring data between host and device, but since the two hardware share a common memory space, and the memory bandwidths are in order of tens or hundreds of gigabytes, it is unlikely that the data transfer between hardware has a large impact here, especially because the required data is only 232 bytes, per call.

7.4 Impact of Implementation Details

For a sparse, dynamic system, such as in this study, a uniform dissection of the whole simulation space is not an ideal solution. Also, as stated by Ericson (2004), successfully im-

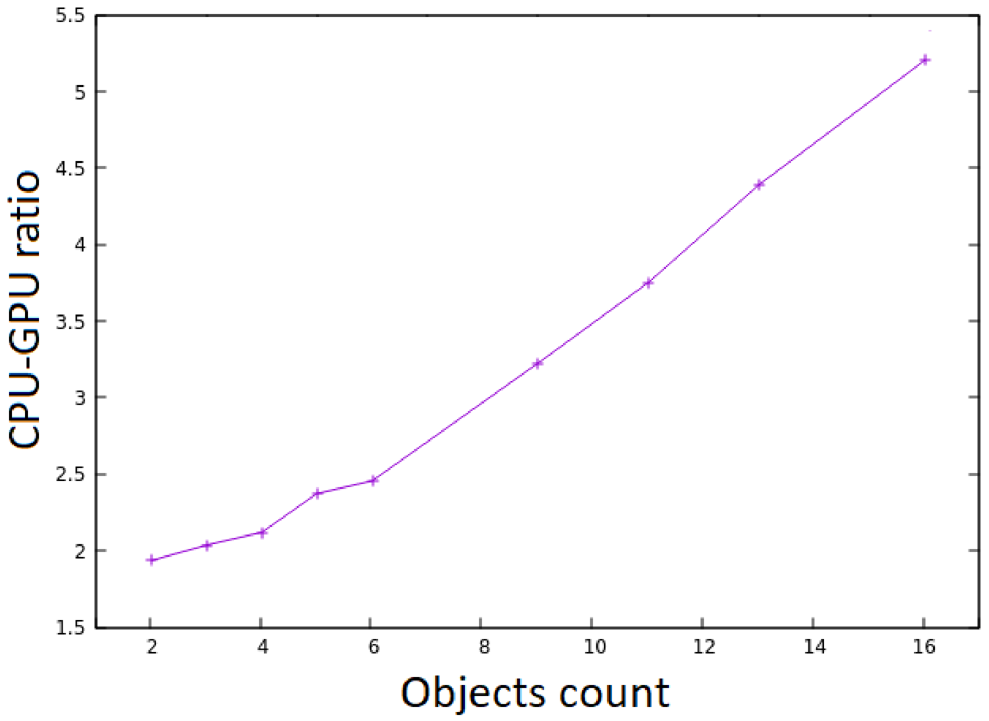

Figure 11. Proportional data acquired from single-core experiment. Within the limits of the utilized dataset, the proportional decrease of performance with an increasing object count has a linear behavior.

plemented acceleration structures should give rise to asymptotic behavior, with respect to an increasing object count. However, the acquired results appear to behave exponentially. This may be because of a wrong choice of the space partition parameters, such as, where the space dividing plane is situated or how small is the smallest cell that still contains the object. These errors may result in only very few subspaces, where all objects are contained, effectively making the broad-phase behave similarly to two nested loops. Also, octree is more often used for evenly spaced structures such as voxels or pixels as Laine and Karras (2010) have done, or for a static scene, such as shown by Revelles, Ureña, and Lastra (2000). Chapter 3.1.2 contains preliminary information and references to broad-phase collision detection methods that should be noted. The other methods, *BVH*, *k-d tree* and *sweep and prune* should probably all be considered before octree in such case as here. Readers new to the subject should see Karras (2012) for GPU implementations of *BVH*, and Baraff (1992) for *sweep and prune*.

Device side SV-GJK executions designed to be performed for two-object sets per thread block. That is, every single object pair interaction would be queried in parallel. The studied sub-algorithm is implemented so that minimal divergence should take place. Depending on the relative orientation of queried objects, threads may be executing different paths concurrently, resulting in separate function calls. Also, the implementation takes into accounting that caching occurs mostly *in situ*. That is, most of the data manipulation is focusing on data declared on the host side, before device execution, where the query data is prepared in an array. Thus, nearby threads will most likely operate on the data in adjacent memory locations, resulting in minimal data divergence. The effect of these factors is difficult to estimate because of the lack of metrics, and the causes affecting the obtained results are most probably stemming from other sources such as, sub-optimal implementation choices.

A notion should be made, that since the problem domain is well known, it's possible that the benefits of the OOP, such as system maintanability and extendability, may be outweighted by the performance gain provided by the procedural or the functional approach.

7.5 Parallellism

A couple of different procedures were attempted to succeed in multi-core experiments. Firstly, for single-core experiment, the allocation worked only with `cudaHostAlloc` (NVIDIA 2019):

```
void checkCollision(// args //)
{
    body *d_bodyA, *bodyA = (bdGPU *)malloc(sizeof(bdGPU));
    cudaHostAlloc(&d_bodyA, sizeof(bdGPU), cudaHostAllocDefault);
    cudaError error = cudaHostGetDevicePointer(&d_bodyA, bodyA, 0);
    if (error != cudaSuccess)
        exit(-1);
//... init / handle host data
//... pass device pointers to device.
// free data
},
```

which was also attempted for the multi-core simulation. Another two schemes were to use `cudaMallocManaged` and also allocating host and device memory separately, and copying data back and forth with `cudaMalloc` and `cudaMemcpy`, as described in 4.4.2. However either the deallocation failed completely with error code cudaIllegalMemoryAddress, which occurred latest at `cudaHostRelease`.

8 Conclusion

Here, the improved GJK algorithm, or SV-GJK, was reviewed. Compared to its predecessors, it has some advantages, that stem from mathematical properties of chosen techniques, such as barycentric coordinates and projections to lower dimensional spaces. Also, the principles of physics engines in video games were described. The reader was also introduced to CUDA and GPGPU. These concepts were used in practical experimentation by implementing simulation software, which applied both CPU and GPU hardware for running SV-GJK as a collision detection algorithm. The hope was to acquire generally applicable results about parallelism of SV-GJK, and see how modern CPU and GPU compare in a benchmark test.

The experiments were conducted by devising a set of well-defined use cases, and measuring the computation time with respect to number of colliding objects. Single-core CPU vs GPU was only successful one, and it shows that for a subsequent execution a single Intel i5 (7th gen.) core outperforms a single-precision FPU on GPU, in terms of handled contacts per second, being 2-5 times more performant, with applied test data, depending on the number of potential colliding objects used in the simulation.

In overall, the work could work as an initial spark for more formal scientific publication. For those who hope to learn about making these experiments, or begin conducting a larger scale research on the subject, a couple of factors are advised to take into careful consideration:

(i) Getting the collision detection algorithm to work properly GPU can be more tedious than in CPU. Pre-assessing the compatibility between the SDK and IDE, compiler and OS, is advised. Alternatively, a more general approach could be to use computation shaders, that are currently supported by most graphics cards, and are independent of the OS or programming language used for the rest of the simulation.

(ii) Test the integration between CPU and GPU implementations and the required tools. The usage of built-in profiling tools etc., are provided by vendors or can be found open-source.

(iii) Careful planning of code design and requirements of implementation should be taken in to account in advance of the implementation. Also, the programming paradigm can indirectly affect the performance. Data-driven-design, functional or procedural features should be preferred over OOP. Here, the problem domain is restricted and fully known, and extensibility and maintainability are less crucial than performance.

Literature

Baraff, David. 1992. “Dynamic Simulation of Non-Penetrating Rigid Bodies”. PhD thesis, Cornell University, Computer Science Department.

Baraff, David, and Andrew Witkin. 1997. *Physically Based Modeling: Principles and Practice.* "https://www.cs.cmu.edu/~baraff/sigcourse/". SIGGRAPH 97 Course notes.

Bender, Jan, Matthias Müller, and Miles Macklin. 2017. “A Survey on Position Based Dynamics”. In *EG 2017 - Tutorials.*

Bentley, Jon Louis. 1975. “Multidimensional binary search trees used for associative searching.” *Communications of the ACM.* 18 (9).

Bergen, Gino van den. 1999. “A fast and robust GJK implementation for collision detection of convex objects.” *Journal of Graphics Tools* 4 (2): 7–25.

———. 2001. “Proximity Queries and Penetration Depth Computation on 3D Game Objects”. GDC.

———. 2003. *Collisions Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers.

———. 2004. “Ray Casting against General Convex Objects with Application to Continuous Collision Detection”. unpublished manuscript.

Bergstra, James, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, Ian Goodfellow, Arnaud Bergeron, Yoshua Bengio, and Pack Kaelbling. 2011. *Theano: Deep Learning on GPUs with Python.*

Billeter, Markus, Erik Sintorn, and Ulf Assarsson. 2010. “Real Time Volumetric Shadows Using Polygonal Light Volumes”. In *Proceedings of the Conference on High Performance Graphics,* 39–45. HPG.

Böhm, Christian, Robert Noll, Claudia Plant, Bianca Wackersreuther, and Andrew Zherdin. 2009. *Data Mining Using Graphics Processing Units - Transactions on Large-Scale Data- and Knowledge-Centered Systems I.* Edited by Abdelkade Hameurlain, Josef Küng, and Roland Wagner. 63–90. Berlin, Heidelberg: Springer Berlin Heidelberg.

Boubekeur, Tamy, and Marc Alexa. 2008. "Phong Tessellation". *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2008)* 27 (5).

Boyd, Stephen P., and Lieven Vandenberghe. 2004. *Convex Optimization.* 27–29. ISBN 978-0-521-83378-3. Cambridge University Press.

Brown, Russel A. 2015. "Building a balanced k-d tree in O(kn log n) time." *Journal of Computer Graphics Techniques.* 4 (1): 50–68.

Catto, Erin. 2007. *Box2D - A 2D Physics Engine for Games.* `"https://box2d.org/about/"`.

Cheng, Wangzhao, Fangyu Zheng, Wuqiong Pan, Jingqiang Lin, Huorong Li, and Bingyu Li. 2018. *High-Performance Symmetric Cryptography Server with GPU Acceleration.* 529–540. ISBN: 978-3-319-89499-7.

Cohen, Jonathan D., Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. 1995. "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-scale Environments". In *Proceedings of the 1995 Symposium on Interactive 3D Graphics,* 189–ff. I3D '95. ACM.

cuSOLVER documentation. 2007. `"https://docs.nvidia.com/cuda/cusolver/index.html"`.

Eigen documentation. 2009. `http://eigen.tuxfamily.org/index.php?title=Main_Page"`.

Ericson, Christer. 2004. *Real-Time Collision Detection.* Boca Raton, FL, USA: CRC Press, Inc.

Firth, Paul. 2011. *Speculative contacts – an continuous collision engine approach part 1.* `"https://wildbunny.co.uk/blog/2011/03/25/speculative-contacts-an-continuous-collision-engine-approach-part-1/"`.

Garg, Adhesh, Diwanshi Gupta, Parimi Praveen Sahadev, and Sanjay Saxena. 2019. “Comprehensive Analysis of the Uses of GPU and CUDA in Soft-Computing Techniques”. In *2019 6th International Conference on Signal Processing and Integrated Networks (SPIN),* 584–589.

Giang, Thanh, Gareth Bradshaw, and Carol O’ Sullivan. 2003. *Complementarity Based Multiple Point Collision Resolution.* Fourth Irish Workshop on Computer Graphics. Conference paper.

Gilbert, Elmer G., Daniel W. Johnson, and S. Sathiya Keerthi. 1988. “A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space.” *IEEE Trans. Robotics and Automation* 4:193–203.

Gottschalk, S., M. C. Lin, and D. Manocha. 1996. “OBBTree: A Hierarchical Structure for Rapid Interference Detection”. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques,* 171–180. SIGGRAPH ’96.

Gregory, Jason. 2009. *Game Engine Architecture 1st Ed.* A K Peters, Ltd. , 830 p.

Gu, Yan, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. *Efficient BVH Construction via Approximate Agglomerative Clustering.* HPG.

Hook, Timothy Van. 1995. “High performance low cost video game system with coprocessor providing high speed efficient 3D graphics and digital audio signal processing”. *US6239810B1.*

Hornus, Samuel. 2017. “Detecting the intersection of two convex shapes by searching on the 2-sphere”. *Computer-Aided Design* 90:71–83. doi:`https://doi.org/10.1016/j.cad.2017.05.009`.

Hubbard, Philip M. 1996. “Approximating Polyhedra with Spheres for Time-critical Collision Detection”. *ACM Trans. Graph.* 15 (3): 179–210.

Implementing GJK. 2006. `"https://caseymuratori.com/blog_0003"`.

JitterPhysics - engine Readme. `"https://github.com/mattleibow/jitterphysics"`.

Karras, Tero. 2012. *Thinking Parallel, Part I: Collision Detection on the GPU.* "https://devblogs.nvidia.com/thinking-parallel-part-i-collision-detection-gpu/".

Khronos Launches Heterogeneous Computing Initiative. 2008. "https://web.archive.org/web/20080620123431/http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/".

Laine, Samuli, and Tero Karras. 2010. "Efficient Sparse Voxel Octrees – Analysis, Extensions and Implementation". NVIDIA Research.

LAPACK documentation. 1992. "http://www.netlib.org/lapack/".

Lin, Ming Chieh. 1993. "Efficient Collision Detection for Animation and Robotics". PhD thesis.

Linahan, Jeff. 2015. "A Geometric Interpretation of the Boolean Gilbert-Johnson-Keerthi Algorithm". *arXiv* abs/1505.07873.

Luebke, David. 2008. "CUDA: Scalable parallel programming for high-performance scientific computing". In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro,* 836–838.

Meagher, Donald. 1980. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer.* Technical Report IPL-TR-80-111. Rensselaer Polytechnic Institute.

Millington, Ian. 2010. *Game Physics Engine Development, Second Edition: How to Build a Robust Commercial-Grade Physics Engine for Your Game.* 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123819768, 9780123819765.

Montanari, Mattia, Nik Petrinic, and Ettore Barbieri. 2017. "Improving the GJK Algorithm for Faster and More Reliable Distance Queries Between Convex Objects." *ACM Trans. Graph.* 36 (30): 1–17.

Morales, Jose Luis, Jorge Nocedal, and Mikhail Smelyanskiy. 2008. *Numerische Mathematik* 111:251–266.

NVIDIA. 2016a. *NVIDIA GeForce GTX 1060 Full Specifications Detailed.* `https://wccftech.com/nvidia-geforce-gtx-1060-final-specifications/`.

———. 2016b. *NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU.* Technical report.

———. 2019. *CUDA Toolkit Documentation.* `https://docs.nvidia.com/cuda/`.

NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256. 1999. `"https://www.nvidia.com/object/IO_20020111_5424.html"`.

Nystron, Robert. 2014. *Game Programming Patterns.* Genever Benning; 1st Ed., 354 p.

Ong, Chong Jin, and Elmer G. Gilbert. 1997. "The Gilbert-Johnson-Keerthi distance algorithm: a fast version for incremental motions". In *Proceedings of International Conference on Robotics and Automation,* 2:1183–1189.

OpenGL 2.0: OpenGl version history, Khronos. 2004. `"https://www.khronos.org/opengl/wiki/History_of_OpenGL#OpenGL_2.0_.282004.29"`.

Pabst, Simon, Artur Koch, and Wolfgang Straßer. 2010. "Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces". *Computer Graphics Forum* 29 (5): 1605–1612.

Pan, Jia, and Dinesh Manocha. 2012. "GPU-based parallel collision detection for fast motion planning". *The International Journal of Robotics Research* 31 (2): 187–200.

PhysX SDK documentation. 2001. `"https://developer.nvidia.com/physx-sdk"`.

Ravishankar, Chinya V., and James R. Goodman. 1983. "Cache implementation for multiple microprocessors", Conference: Sponsored by IEEE.

Revelles, J., C. Ureña, and M. Lastra. 2000. "An Efficient Parametric Algorithm for Octree Traversal". In *Journal of WSCG,* 212–219.

Richards, Tonge, Feodor Benevolenski, and Andrey Voroshilov. 2012. "Mass Splitting for Jitter-Free Parallel Rigid Body Simulation". *ACM Transactions on Graphics* 31 (4): 1–8.

Schauer, Johannes, and Andreas Nuchter. 2015. “Collision detection between point clouds using an efficient k-d tree implementation”. *Advanced Engineering Informatics* 29 ().

Snethen, Gary. 2008. “Complex Collision Made Simple”. *Game Programming Gems* 7:165–178.

Stone, John E., James C. Phillips, Peter L. Freddolino, David J. Hardy, and Trabuco Klaus Schulten. 2007. “Accelerating molecular modeling applications with graphics processors”. *J. Comput. Chem:* 10–1002.

Unity3D - Manual, Continuous Collision detection. `"https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html"`.

Verlet, Loup. 1967. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules”. *Phys. Rev.* 159 (1): 98–103.

Wilt, Nicholas. 2013. *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Addison-Wesley Professional, 520 p.

Xenocollide library. 2006. `"https://protogame.readthedocs.io/en/latest/jitter/api/xenocollide.gen.html"`.

Xiao-rong, W., W. Meng, and L. Chun-gui. 2009. “Research on Collision Detection Algorithm Based on AABB”. In *2009 Fifth International Conference on Natural Computation,* 6:422–424.