Collision Detection in Physics Engines

Fadl Akkad

May 2016

Abstract

We explore the use of an elaborate collision detection system; multiple objects detecting when collision will occur with each other, in a digital three-dimensional environment utilising the Bullet physics engine libraries. A significant subject for the construction of a realistic and immersive world. We investigate collision detection, its essential components and its various applications. We create a demonstration program of an interactive environment where multiple objects with differing properties of position and velocity are set on a course for collision. We observe the Bullet libraries functionality and provide proposals on how to best utilise Bullet in regards to crafting a more refined system of object collision detection, catering for improvements such as speed and accuracy to deliver a more effective and efficient method for deployment.

Project Dissertation submitted to the Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

May 6, 2016

Signed: Fadl Akkad

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Computer Science.

May 6, 2016

Signed: Fadl Akkad

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

May 6, 2016

Signed: Fadl Akkad

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

May 6, 2016

Signed: Fadl Akkad

Contents

1 Introduction

Creative and interactive software is constantly evolving and becoming more immersive with each advancement. As time has progressed creative software's graphics have continuously improved and technology has allowed for the introduction of various new and elaborate mechanics. Among such technical refinements include the increasingly realistic simulation of real world physics.

Throughout the creative software industry and its history, many contributions have been made to the advancement of physics simulation, most notably is Erwin Couman's Bullet physics engine which has been used for many applications such as critically acclaimed video games and animated movies. Bullet and many more physics engines have made progressive steps to crafting more immersive experiences particularly by simulating various collision and deformation effects in a digital environment. It goes without saying that this fascinating and intricate aspect of creative software will continue to evolve in the future, but in order to enrich ourselves in this area of study, and thus participate in its development, we must acquaint ourselves with the general concepts and ideas behind physics simulation and thus broaden our understanding of this topic.

One such element of physics engines in particular is realistic collision detection between objects. The goal of this project will be to demonstrate how collision detection occurs using the Bullet library.

Realistic object collision detection is critical if one wants to conduct a more immersive digital environment, of course the way various objects and digital apparatuses interact with each other needs to be seamless so that the human eye can better approve and believe what it sees. Object collision is incredibly important in regards to this. It is the mission of programmers and digital artists alike for their interactive product to be as real to life as possible, or at least to a degree necessary to realise their artistic vision. When two or more objects collide, various factors come into being, for example; the mathematical consequences of the collision such as the subsequent directions and velocities and other properties that objects inherit post collision. Actual detection of inevitable collision between objects in a real time environment is an important factor that we must consider. It is imperative that our environment be enabled with a concise system of algorithms dedicated to an elaborate object collision detection system that is consistently conducted on all objects and general 3D apparatuses in an environment.

For this project; We shall study the phenomena of object collision and how physics engines such as Bullet conducts such operations. Using the Bullet library we will form a interactive 3D environment which we will use to show object collision detection in action and thus use such a program to demonstrate the essentials in our study of object collision detection and its applications.

2 Project Definition

The primary objective of this project will be to assess the Bullet library using computational languages such as OpenGL. In order to better our understanding of physics engines, it is necessary to experience first-hand the workings of the engine and profess in realising the mechanics behind the opportunities provided by the Bullet library with practice. Ultimately the desired result of this project will be to develop and emerge a foundation for comprehending this topic and thus exploring how digital physics simulation may be better advanced in the future.

More concretely, the goal of the project is to practice and assess the Bullet library performance using OpenGL to render digital information, primarily using spheres as the objects of interest and to acquaint ourselves with object collision in a three-dimensional environment. And thus make progressive steps in simulating object collision using various methods, using the variety of functions that the library has to offer to discover more efficient and advantageous ways of simulating object collision.

The key area of exploration is the detection of collision between 3D objects and how we may be able to generate a better and more efficient performance with collision detection. Beginning with the initial idea that graphics are rendered using geometric primitives, primarily in the form of triangles. A main goal of the project will be to create a working demo of object collision in action, and to demonstrate the proficiency of object collision in progressive steps. We will study and observe how the Bullet library operates in regards to object collision detection and also revise the essentials of digital object collision detection in general such as collision between two triangle primitives and so called 'bounding box' volumes used in a 'k-dimensional' hierarchy throughout the 3D scene. We will also explore formulating the most beneficial and advantageous collision detection algorithm.

The end result of this project will be to derive a interactive demo were we can see the Bullet library in action, where collision detection occurs between multiple objects. This demonstration will focus predominantly on multiple spherical objects being deployed in the scene and issued different velocities and directions.

3 Research on Object Collision Detection

We present the following research and required contextual knowledge on the subject of digital object collision detection for our study:

3.1 Related Work

This subject material is one that has fascinated many researchers and developers a like, it is important to familiarise ourselves with prior research and reports on this topic. A significant work on this topic may be a paper in 1988 written by Matthew Moore and Jane Wilhelms [MW88] which speaks about the initial problem of objects interpenetrating and the demand for a system to be put in place to better handle a two-step solution, detecting the problem and then responding to it. It is imperative that this be kept in mind in the development and testing of physics engines. Much research has been done on the greater subject of the advancement of physics engines, we may look toward Adrian Boeing and Thomas Braunl [BB07] in their paper which presents their views on the various aspects of physics engines and more importantly the consistent and ever demanding need for more elaborate techniques to simulate more realistic object collision detection.

There are various forms of object collision detection, from a triangle primitive with another to a bounding box with another. Further in this document we will exam the following prior researcher's findings in the area of object collision detection. Tomas Moller [Mol97] discusses at length the definitive collision detection algorithm between two triangle primitives, Moller's paper and also the research and contributions of Gottschalk [GLM96], Hubbard [Hub96] and Klosowski [Klo+98] will prove valuable when reflecting on the need for more efficient ways to register collision between primitives based on their research with 'tree' and 'sphere' hierarchies. When it comes to the collision detection between a box and sphere James Arvo [Arv90] in his 1990 work had crafted a refined and thorough method of carrying out such a computational procedure, however other researchers such as Thomas Larsson and company [LAL07] have also provided their own improvements upon Arvo's methods.

A significant paper on the subject of collision between spheres is the 1997 paper 'Fast Collision Detection among Multiple Moving Spheres' by Dong Jin Kim et al. [KGS97] The paper discusses at length various methods of ensuring collision detection between sphere objects is at its most accurate and fastest. One such method includes splitting the space into a set of uniform subspaces of cell structure, keeping track of the path of every sphere and the list of spheres intersecting each subspace. These paths and lists are frequently changed every time there is a collision or when a sphere moves into a different subspace. This method is further substantiate evidence that a most effective and popular way to deploy collision detection is by methodically organising the 3D space into hierarchical sections.

3.2 Understanding Physics Engines and Computational Languages

Today, the development of creative software has become more accessible to varying degrees of developers and researches alike from novices to experts. Development tools such as physics engines have become a sub culture of their own as they have acted as middleware, and in modern times there has been communities dedicated to creating and sharing creative content using such tools. As Boeing and Braunl [BB07] show, the range of physics engines available to the aspiring developer is abundant, and they provide a range of components and features, most notably for our case study, the simulation of object collision.

For this project we will use the OpenGL computational language to render and examine scenes that operate using the Bullet physics engine's library. The Bullet physics engine focuses on the aspects of collision and specialises in the realistic simulation of both soft and rigid body transformation. Bullet provides continuous opportunities for collision detection for multiple shapes, whether it be a triangle, sphere, cone or box.

OpenGL is a language and application programming interface purely concerned with rendering that maps graphical information to the machine's GPU (graphics processor unit) and is independent of any one operating system, allowing for flexibility across multiple platforms. OpenGL like many computational languages provides a range of extensions and tool-kits which will help render select shape and objects specific to the situation desired.

It is evident that one of the key aspects of this project is to consider and evaluate the behaviour of objects during and after collision has occurred, the Bullet library can be explored and used to create a more seamless and immersive simulation of real world physics. Using Bullet we may calculate attributes of an object's sequential steps of movement, depending on various factors such as the detection and point of collision, and how the object in question would react when it comes to weight and speed among other characteristics.

In summary the Bullet physics engine and OpenGL both provide the tools needed to render and calculate the simulation of object collision in a three-dimensional environment. Such software/middleware has been used with critical acclaim to create many immersive forms of creative software and will be used in our investigation.

3.3 Constructing and Rendering a 3D Environment

The three-dimensional space that needs to be created does not need an elaborate environment for our case study, however many tests will be performed in this space, with various different objects, and scenarios for collision. We will be using the 'C++' programming language for our approach to the OpenGL API and thus rendering our environment. It is worth explaining how a 3D space will generally be created not just by OpenGL or any other language with a similar function but also the general concept of the procedure.

Primitives are usually formed in the shape of triangles or on a more basic level a set of vertices and straight lines. In principle all shapes including rounded shapes and other more complex variants are formed using such primitives. However in practice a problem emerges when we use lots of these vertices and triangles in constructing our shape/environment, immediately we realise that this leads to much expense for our machine. We find that we have exhausted a great portion of memory in order to construct our environment, this not only slows down operations of our machine, but also increases the number of calculations needed in general which will prove to be a disadvantage when manipulating our environment/shapes for the purpose of object collision. This will of course greatly affect our ability to detect collisions. There are many ways one can overcome this situation, but these drawbacks will have to be kept in mind for later in this document when I talk about bounding boxes and their associated hierarchy and its advantages in reducing calculations and increasing overall efficiency by grouping common primitives.

When our shape(s) or models have been constructed, the rendering process (generating the final image in its desired form) may be accomplished using a variety of methods. If we were to establish a more complex scene and environment we may use a more elaborate method such as ray tracing, where 'rays' are deployed into the scene to make contact and intersect with primitives to collect data about the scene and determine how the scene may best be rendered with respect to attributing factors such as reflections, refractions and shadows. Rasterization is a common method of rendering which includes using matrices to scale, translate and rotate the model for desired adjustments and then colouring the pixel according to provided information whilst focusing on the model's place within the general 3D space. We may assume that our scene and models in OpenGL will be generated using the direction of rasterization as I have described or similar by specifying various vertices and their relation to each other.

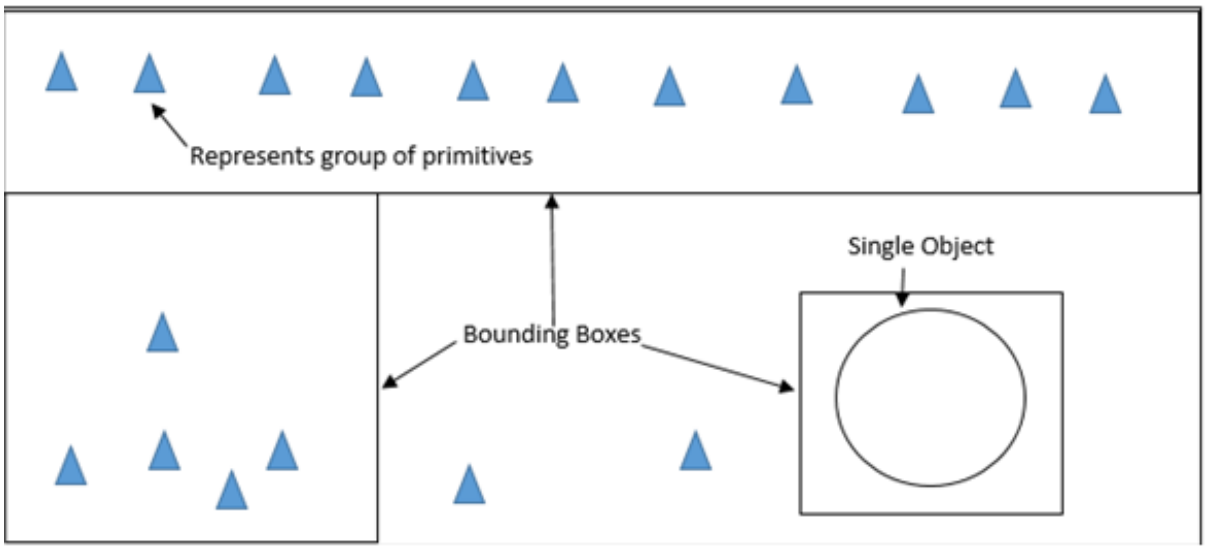
Ultimately our case study is not concerned with elaborate environments, but instead a simplistic 3D space where collision detection may be tested, however we must keep the process of rendering in mind when we use OpenGL to construct our scene, as computational power will be needed for assessing efficient detection algorithms and object responses to collision.

3.4 General Collision Detection Methods

A key factor of object collision detection is that objects are comprised of many primitives.

A common method used when constructing and rendering 3D shapes and environments is the act of segregating and grouping primitives based on the common area that they share within the space. In the process of ray tracing, bounding boxes (or bounding volumes) are used to collect primitives together and have them act as a single unitary collective instead of separate and independent agents. When a ray intersects with the box, all primitives are registered. When an object collides with the 'bounding box' or segregated area of the entire space, it will be as though the object is interacting with all primitives in that area. Ultimately we may have many of the boxes in the scene, not just in designated areas of common primitives but also objects themselves, helping to reduce calculations as illustrated in Diagram A.

Diagram A

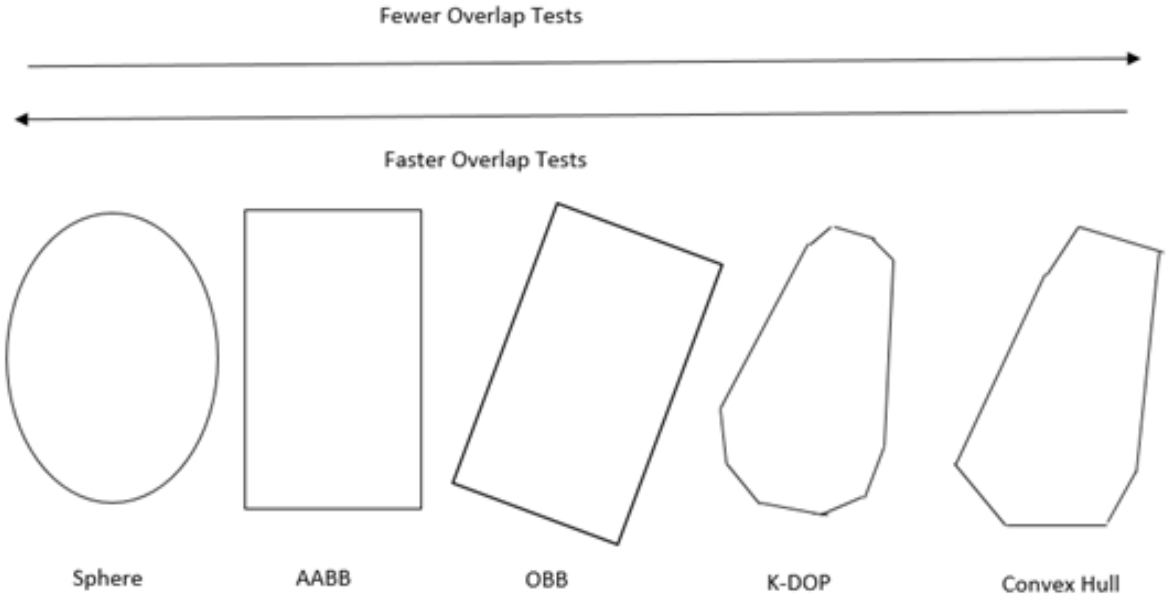


As shown above, bounding boxes are focused on collecting a large group of primitives together, in this case based on region and their proximity to each other. When the sphere (or rather its box) comes into contact with a contemporary box, it is making contact with all primitives therein, thus when two bounding boxes do not intersect, the contained objects/primitives cannot collide. Of course these boxed primitives may share embedded relationships with each other may form a hierarchy style 'k' – dimensional tree, thus reducing complexities required in order for the scene to operate. This method helps to structure a scene to have a more seamless operation of physics simulation and help derive detection algorithms.

When it comes to the actual process of object collision detection, as discussed before, there has been much research in regards to the nature of this topic and how collision detection may be better comprehended and thus how physics engines may better utilise collision detection algorithms.

Gottschalk and company [GLM96] speak at length about bounding volumes or or *oriented bounding boxes* (OBB's) as opposed to *Axis-Aligned Bounding Boxes* (ABB's). OBB's are geometrically situated in a more advantageous position or stance that provide a better opportunity for detecting object collision. A tighter and more seamless fit around the object implies better rates of detection, however OBB's are more polygon central, and there is a significant disadvantage when it comes to curved surfaces. Klosowski [Klo+98] and company move further along the line of thinking that tighter bounding volumes are the way forward, using what is called discrete-orientation polytopes in favour of the idea that there will be fewer overlap tests, thus containing some degree of accuracy. Hubbard [Hub96] however proposes using *time-critical computing* which trades accuracy for speed, Hubbard's detection algorithms checks for collisions between successfully tighter approximations (bounded volumes) to the objects real surface. Hubbard argues that although this approach may lead to some inaccuracy, small inaccuracies are acceptable in many situations. Overall we may examine the options available for bounding volumes in Diagram B.

Diagram B



Moller [Mol97], Larsson and company [LAL07] and Arvo [Arv90] all present speedy and efficient forms of detection algorithms when it comes to primitives, spheres and boxes, and ultimately the demo for our case study will culminate in having a variety volumes intersect, noting the primitives that have been registered during contact. This implies that the definitive algorithm may be based on the intersection between two boxes or boxed object with a segregated area of space.

It is worth pointing out that methods for intersection and thus also calculating object responses have been constantly revised throughout the history of research in this topic, when programming and calculating for our case study it is imperative to consider that there is always a opportunity to grasp a opportune moment to implement for efficiency and accuracy depending on the situations we may encounter when experimenting with and testing the Bullet library.

3.5 Formulae and Calculations for Sphere - Sphere Collision Detection

Much research has been made in the general area of understanding and applying the necessary calculations and algorithms needed for the program to operate with fully functional object collision.

Pairwise Collision Detection Algorithm

We use the Bullet library to implement the equation needed to detect an imminent collision between two spheres. Each object has a minimum and maximum value for the x, y and z coordinates. Within two designated values along the co-ordinate we compute the number of overlaps and the biggest distance. The best co-ordinate would be equal to the one with the minimum number of overlaps and the maximum distance.

Once this has been accomplished, we may perform pairwise collision between two spheres:

$(x1–x)^2 + (y1–y)^2 + (z1–z)^2 < (r + r1)^2$ (if true, a collision is detected)

Subsequent Actions to be Performed Due to Object Collision

It is imperative to acknowledge the consequence of two objects colliding with each other, in the various possible directions and speeds and the need for realism of that aspect. It is critical therefore to implement necessary equations needed to simulate objects changing their behaviour in a realistic fashion, such as 'bouncing' in a different corresponding angle due to collision.

We must first consider that when the two vertices are about to intersect (the vertices that the spheres are moving along in their course) a instance of a triangle is created, it is this triangle which allows us to calculate the correct angle our object should be changing direction towards. This is done using a vector surface normal together with the radius of the sphere to adjust the x, y and z co-ordinates to the new respective set of co-ordinates.

Accommodating for Inelastic Collision

Additionally, the adjustment of the speed of the object due to collision must also be acknowledged and considered when using the Bullet engine. Discovering the new velocity of our object is important for realistic effect and the principles of inelastic collision can help us adjust our program to accommodate this. Ideally our spheres will have equal mass.

$((v1m1) + (v2m2))/(m1 + m2)$

A focal point of this investigation will be exploring how the Bullet library performs these operations.

4 Project Development

Since initiating this project, it has gone through several evolutions and changes. Specifically the methodology of how to design the demonstration, and its inherent functionality. It was decided from the very beginning that the demonstration would be, in essence a 3D scene, with multiple spherical objects set on a course for collision, this would allow me to analyse the Bullet library and investigate its workings, and how it allows for realistic object collision detection.

The project doubled in its purpose as a learning experience for this author. Throughout the course of the project, this author had developed significant skills with the 'C++' programming language as well as the OpenGL API. Initially the goal was to create a small enclosed 'room' style space, where objects could be deployed with randomly generated positions, directions and velocities. Eventually I decided upon having a more open space world with large x, y and z dimensions, focusing computational power on the efforts of object collision detection. The program may generate lots of sphere objects in the x, y and z plane and include variation of sizes for the spheres. It became apparent that my scene would have to be interactive by the user in order to observe or focus on certain parts of the scene. It was because of this that we use the Simple Direct Media Layer library or SDL to have a movable and interactive camera that response to user input such as key presses and mouse movements.

For this project this author used educational resources dedicated to the C++ language, specifically 'The C++ Programming Language' by Bjarne Stroustrup [Str13] and also 'Computer Graphics with OpenGL' by Hearn, Baker and Carithers [HBC10].

The final demonstration that we derive is a product that fully enables us to explore the Bullet library and how its components apply to collision detection. Further in this project we will analyse the Bullet library in action utilising our demonstration and we will develop a more coherent understanding of the implementation of such elaborate physics simulations.

5 Demonstration Design

We present the essential fabric of the demonstration including its logical and aesthetic components.

5.1 Establishing Bullet Components

We establish the connection of the program to the Bullet library and its functionality, importing the major super class "btBulletDynamicsCommon.h".

```
// Establish Bullet functionality located in libraries for the scene.
// 'DynamicsWorld' is a major super class in the library.
btDynamicsWorld* world;
```

We observe that our program traverses the Bullet library and its hierarchical structure. First we instruct our program that Bullet is to operate in the world. The three significant libraries used for this project are 'BulletCollision','BulletDynamics' and 'LinearMath'. These provide the necessary and critical functions needed including establishing Bullet to have an authority and presence in our scene.

```
// Dispatches calculations for overlapping pairs,
// thus when pairwise collision occurs between the spheres,
// this function will calculate contact points and derive the
// necessary consequent algorithm.
btDispatcher* dispatcher;

// Processes what collision algorithm to use based on
// collected information including contact points and shape
// information.
btCollisionConfiguration* collisionConfig;
```

Later in this report we will analyse the actual Bullet functions in further detail, but it is important that we understand how the Bullet library accesses our demonstration. As discussed, it is a general concept in collision detection that various algorithms collaborate together to provide efficient systems of collision detection. Notice that the Bullet library contains many potentials for implementing such algorithms, we can see with the dispatcher function that the general algorithm for sphere-sphere collision detection that we have discussed is handled by the Bullet library. Bullet understands which is the best pairwise algorithm to perform based on preset information of the scene including contact points (manifold points) and recognising Bullet shapes such as PROXY - SPHERE. The dispatcher and collision configuration functions traverse the library with this information to return the corresponding algorithm to be performed.

```
// Function which examines the scene to assign priority to more populated
// segments of the scene, and establishes aabb(s) or axis-aligned bounding
// boxes for faster overlap tests across that particular segment.
btBroadphaseInterface* broadphase;
```

Bullet not only understands how and when to perform pairwise collision but also the previously discussed axis-aligned bounding boxes or AABBs. Broadphase is a function which constantly studies the scene and prioritises where more tests need to be performed at any given time, for example a segment with more spheres than other segments than further super imposed bounding box algorithms will be performed for that segment.

```
// Function which solves collisions, apply forces, impulses and provides new
// behaviour for the objects post collision.
btConstraintSolver* solver;
```

The constraint solver function provides additional parameters to be considered including any additional impulse or force constrictions applied to the objects, this is a critical function for when the object do collide with each other and rebound or generally react in some way.

```
/*
Construct 'objects' recognised by Bullet that collaborate with each OpenGL constructed sphere.
Each object has its own unique identifier, aesthetic information and for the objective of collision
a boolean that will be used to recognise when collision has occurred.
*/
struct bulletObject {
        int id;
        float r, g, b;
        bool hit;
        btRigidBody* body;
        bulletObject(btRigidBody* b, int i, float r0, float g0, float b0) :
        body(b), id(i), r(r0), g(g0), b(b0), hit(false) {}
};

// For each Bullet object deployed in the scene, provide them with
// a sequence container so that the objects can be stored.
std::vector<bulletObject*> bodies;
```

Our Bullet object is crafted with its logical and arithmetic components ready to receive further characteristics of behaviour and also allows us to easily super impose OpenGL aesthetic features on to it so that we can visualise the object.

The Bullet object is categorised into a vector, of which one shape is a body in a series of bodies that are each treated as unique instances of spheres, this vector will allow us to assign collision flags which will be discussed in detail shortly. Each sphere is assigned a collision flag which enables two objects, each with its own flag to collaborate in order to detect collision.

5.2 Deployment of Spheres and their Properties

The following is code that instructs the generation of spheres and their randomised properties.

```
for (int i = 1; i <= sphereCount; i++) {
    // Assign whether the sphere shall traverse the x, y or z axis.
        direction = rand() % 4 + 1;
        // Assign the x co-ordinate of the spheres position.
        int randomX = rand() % 200 + 1;
        xLocation = randomX;
        // Assign the y co-ordinate of the spheres position.
        int randomY = rand() % 100 + 1;
        yLocation = randomY;
        // Assign the z co-ordinate of the spheres position.
        int randomZ = rand() % 200 + 1;
        zLocation = randomZ;
        // Assign the size of the radius - very small variations.
        int randomRadius = rand() % 2 + 1;
if (threeD == true) {
                addSphere(randomRadius, xLocation, yLocation, zLocation, 1.0);
        }
        else {
                addSphere(randomRadius, xLocation, yLocation, 0.0, 1.0);
        }
        // NOTE: 'zLocation will be 0.0 should the user prefer to have the spheres
        // operate on a 2D plane in our scene.
}
```

Before a sphere can properly be added into the scene we give the sphere its properties that it shall deploy with. We generate a random number for which axis the sphere will move on, called 'direction' because it is the axis that the sphere will traverse after deployment, forward or backwards depending on a later variable called 'speed'. The location of deployment for each sphere is a random generation of all 3 co-ordinates x, y and z. The radius is also given.

The user as a choice between aligning all spheres two dimensionally (ignoring the z axis) or including the z axis.

The following is the method by which a sphere is included into the scene.

```
/*
Method which deploys a sphere into the world, with provided information including the radius,
its position of deployment and its initial mass.
*/
btRigidBody* addSphere(float rad, float x, float y, float z, float mass)
{
    ...
    ...
        // Provide a random number for the speed of the object,
        // depending on the user's choice.
        // NOTE: Negative 'speed' implies moving backwards on axis.
        if (fast == true) {
                speed = rand() % 101 + -100;
        }
        else {
                speed = rand() % 4 + -3;
        }
        if (direction == 1) {
            // If the sphere was assigned to the X axis,
            // instill movement on the X axis.
                body->setLinearVelocity(btVector3(speed, 0, 0));
        }
        else if (direction == 2) {
            // If the sphere was assigned to the Y axis,
            // instill movement on the Y axis.
                body->setLinearVelocity(btVector3(0, speed, 0));
        }
        else if (direction == 3) {
            // If the sphere was assigned to the Z axis,
                // instill movement on the Z axis.
                body->setLinearVelocity(btVector3(0, 0, speed));
        }
        // Set a 'collision flag' to the sphere, a unique Bullet function to ensure
        // Bullet is able to recognize contact with another shape that also has a
        // collision flag.
        body->setCollisionFlags(body->getCollisionFlags()
        | btCollisionObject::CF_CUSTOM_MATERIAL_CALLBACK);
        body->getMotionState()->getWorldTransform(t);
        // Store the sphere in a vector sequence container
        // complete with a unique identifier for the purpose
        // of recognizing collision.
        bodies.push_back(new bulletObject(body,1,1.0,0.0,0.0));
        body->setUserPointer(bodies[bodies.size() - 1]);
        return body;
}
```

'btRigidBody' is an inherited class of 'btCollisionObject'. The structure of this class allows for recognition of contact between objects, utilising various functions of that particular library.

I decided during the development of the demonstration that the user should be able to choose whether or not the velocity of the spheres should be fast or slow, in combination with axial placement to provide them customisation for observation of collision detection. The 'speed' is provided to the axial placement that was assigned to the sphere, this also helps provide the general direction that the sphere will move or drift towards. We shall examine 'collision flags' which enable detection to occur more closely later in this report.

5.3 Utilising OpenGL to Provide Graphical Correlation

We explain the following method which acquires information that will help ensure consistent aesthetic visuals, the method renders graphical visuals in co-ordination with Bullet mathematics. The OpenGL shape is super imposed on the Bullet 'shape' information to allow us to observe Bullet visually.

```
    void renderSphere(bulletObject* bulletobj)
{
        btRigidBody* sphere = bulletobj->body;
        if (sphere->getCollisionShape()->getShapeType() != SPHERE_SHAPE_PROXYTYPE)
                return;
        // Acquire information that will help ensure consistent aesthetic visuals.
        glColor3f(bulletobj->r,bulletobj->g, bulletobj->b);
        float r = ((btSphereShape*)sphere->getCollisionShape())->getRadius();
        btTransform t;
        sphere->getMotionState()->getWorldTransform(t);
        float mat[16];
        // OpenGL matrix stores the rotation and orientation.
        t.getOpenGLMatrix(mat);
        glPushMatrix();
        // Multiplying the current matrix with it moves the object in place.
        // Ensure consistent collaboration between Bullet and OpenGL.
        glMultMatrixf(mat);
        // Finalize quadric object.
        gluSphere(quad, r, 20, 20);
        glPopMatrix();
}
```

Both Bullet and OpenGL are in constant collaboration with each other, in essence, graphics and physics. In actuality, there are two shapes super-imposed upon one another, we have observed the construction of the Bullet rigid body with mathematical instructions to convey collision contact points. Our demonstration works on a mathematical level, collisions are still detected purely from these constructed Bullet objects due to the algorithms endowed upon them. The OpenGL API as explained before is used to provide graphical visuals, although it must be understood that the OpenGL implementation simply 'chases' or follows the generated Bullet 'shapes'. This is done by multiplying the OpenGL quad shape matrix at points corresponding with the Bullet transform location information or the position of the Bullet shapes position and rotation.

5.4 Ensuring Frequent Courses for Collision

We present the following method used to provide variations of sporadic changes in motion of the spheres.

```
/*
Method to provide variations of impulse forces on the spheres throughout the scene in real time.
*/
void changeDirection(bulletObject* bulletobj)
{
        btRigidBody* sphere = bulletobj->body;
                int change = rand() % 11 + 1;
                if (change == 1) {
                        sphere->applyCentralImpulse(btVector3(0.003, 0, 0));
                }
                else if (change == 2) {
                        sphere->applyCentralImpulse(btVector3(0, 0.003, 0));
                ...
                ...
                else if (change == 12) {
                        sphere->applyCentralImpulse(btVector3(-0.003, -0.003, -0.003));
                }
}
```

This method's purpose is to simply provide a way for their to be consistent variation for the potential of collision during the course of the sphere's main course of direction or velocity. This author decided that it was nominal for the variation to be small and less than one so as not to conflict with the main directive of the velocity of the sphere, or less the spheres would disperse out of the scene much more faster and would be counter intuitive to the objective of prompting occasions for testing collision detection.

5.5 Applying Bullet Functionality to Register Collision Detection

We present the following critical method needed for Bullet to conduct essential algorithms to detect collision.

```
/*
Method responsible for detecting collision between objects, by having two dedicated collision flags
collaberate.
*/
bool collisionDetection(btManifoldPoint& collide, const btCollisionObjectWrapper*
obj1, int id1, int index1, const btCollisionObjectWrapper* obj2, int id2, int index2){
        // The flag is located in the storied body of the collision object, a derivative
        // of the function 'btCollisionObjectWrapper'. A collision flag is composed of unique
        // binary number identifiers that Bullet recognizes, this helps pairwise collision
        // when a test is being conducted either outside or inside an aabb.
        std::cout << ((bulletObject*)obj1->getCollisionObject()->getUserPointer())
        << " Collision Detected! " << ((bulletObject*)obj2->getCollisionObject()
        ->getUserPointer()) << std::endl;
        return false;
}
```

Each object deployed into our scene is equipped with a collision flag. Bullet has a variety of these 'flags' or markers which main purpose is to indicate a certain characteristic about the object that implies something about its level of interactiveness with other objects that either also have or don't have the same flag allotted to them. There exists flags which recognise certain characteristics in objects such as whether they are kinetic for example these flags can help cause desired consequences or implications. For this demonstration our spheres are enabled with a collision flag, when 'manifold points' or contact points come into contact, the flags report to Bullet a collision should occur and the dispatcher and collision configuration traverses algorithms to return. We shall examine Bullet's flags and other functions in more detail further on.

6 Testing

We present the methods of testing our demonstration program and the results as well as their implications.

Collecting User Input

The program query's the user for three major settings preferences that it should implement for the scene. These settings are whether or not the user would refer to have the spheres be deployed on a 2D or 3D plane, as well as the speed of the spheres and the number of spheres to be deployed. The input is collected and applied appropriately and as intended, provided that the user provides the correct input data type. The user is also warned that a high number of spheres could cause heavy exhaustion of computational power.

Generating the Scene

The program proves to be very efficient when generating the scene, the graphical visuals represent the Bullet dynamics and collision points that Bullet has implemented, and I can see that the Bullet collision system is fully functional as intended due to output indicating collisions are detected before actual collision occurs. The spheres motions are consistent with the programmed directives that was previously discussed.

Collision Detection and its Efficiency

The collision detection system itself is efficient and works seemingly well and as expected. When there is a cluster of spheres, output provides a constant series of detections that occurred, indicating that Bullet is providing axis-aligned bounding boxes in heavily clustered areas due to the fast and frequent testing being performed. This is especially noticeable with the more spheres that are populated in the environment. The demonstration itself is of course designed with testing in mind. Each sphere has its own unique identifier and output provides an alert for each detection between two spheres.

We implement testing for a variety of scenes with differing populations of spheres. This allows the user to choose between observing a fast paced and frantic collision scenario or a chance to observe the detection system in a much more contained and focused environment. The demonstration is designed to show Bullet handling collision detection and enables the user to have a guide in their own observation of how physics engines such as Bullet applies and implements such an elaborate physics simulation, especially if they observe the programmed directives of our demonstration and how it utilises the Bullet library.

7 Examination of the Bullet Library

We present a study of the Bullet library and its functions and way of implementation towards our demonstration.

7.1 Constructing Contact Points of Collision Shapes

When it comes to collision detection 'btRigidBody' is a super class that endows upon constructed objects various functions and classes designed to enable the object to behave in the way desired by the programmer. One such class is 'btManifoldPoint'.

```
class btManifoldPoint
        {
                public:
                        btManifoldPoint()
                                :m_userPersistentData(0),
                                m_lateralFrictionInitialized(false),
                m_appliedImpulse(0.f),
                m_appliedImpulseLateral1(0.f),
                                m_appliedImpulseLateral2(0.f),
                                m_contactMotion1(0.f),
                                m_contactMotion2(0.f),
                                m_contactCFM1(0.f),
                                m_contactCFM2(0.f),
                                m_lifeTime(0)
                        {
                        }

                        btManifoldPoint( const btVector3 &pointA, const btVector3 &pointB,
                                        const btVector3 &normal,
                                        btScalar distance ) :
                                        m_localPointA( pointA ),
                                        m_localPointB( pointB ),
                                        m_normalWorldOnB( normal ),
                                        m_distance1( distance ),
                                        m_combinedFriction(btScalar(0.)),
                                        m_combinedRollingFriction(btScalar(0.)),
                                        m_combinedRestitution(btScalar(0.)),
                                        m_userPersistentData(0),
                                        m_lateralFrictionInitialized(false),
                    m_appliedImpulse(0.f),
                    m_appliedImpulseLateral1(0.f),
                                        m_appliedImpulseLateral2(0.f),
                                        m_contactMotion1(0.f),
                                        m_contactMotion2(0.f),
                                        m_contactCFM1(0.f),
                                        m_contactCFM2(0.f),
                                        m_lifeTime(0)
```

As the Bullet library explains; 'ManifoldContactPoint' collects and maintains persistent contact points of all 'rigid bodies'. This is performed by constant tracking of the shape and its traversal of its designated 'vector 3' co-ordinates and all properties that are involved with 'btRigidBody' modified through inheritance/polymorphism.

'btManifoldPoint' also recognises additional properties of the object that govern additional behavioural characteristics.

```
                        btScalar getDistance() const
                        {
                                return m_distance1;
                        }
                        int        getLifeTime() const
                        {
                                return m_lifeTime;
                        }

                        const btVector3& getPositionWorldOnA() const {
                                return m_positionWorldOnA;
//                                return m_positionWorldOnB + m_normalWorldOnB * m_distance1;
                        }

                        const btVector3& getPositionWorldOnB() const
                        {
                                return m_positionWorldOnB;
                        }

                        void        setDistance(btScalar dist)
                        {
                                m_distance1 = dist;
                        }

                        ///this returns the most recent applied impulse,
                        ///to satisfy contact constraints by the constraint solver
                        btScalar        getAppliedImpulse() const
                        {
                                return m_appliedImpulse;
                        }


        };
```

A prominent example of this would be its handling of impulse. In our program we have our spheres endowed with 'central impulse' to provide variation of the behaviour of the motion in spheres. This is done by a collaboration with 'btConstraintSolver' which is a class meant for the purpose of keeping the behaviour of the objects consistent post collision. We can see that Bullet persists in its objective of maintaining an elegant physics system throughout our demonstration by operating multiple processes designed to ensure each object properly emulates desired real world physics characteristics all the way through to termination.

7.2 Collision Flags

As we have discussed earlier, our program provides every sphere with a collision flag. The proper term Bullet uses for the flag we use in our program is 'CF_CUSTOM_MATERIAL_CALLBACK'. This enables the constructed contact points to be able to detect with the algorithm given by the dispatcher inevitable contact points that belong to other spheres. The following is from the class 'btCollisionObject'.

```
enum CollisionFlags
{
        CF_STATIC_OBJECT= 1,
        CF_KINEMATIC_OBJECT= 2,
        CF_NO_CONTACT_RESPONSE = 4,
        CF_CUSTOM_MATERIAL_CALLBACK = 8,//this allows per-triangle material
                                        //(friction/restitution)
        CF_CHARACTER_OBJECT = 16,
        CF_DISABLE_VISUALIZE_OBJECT = 32, //disable debug drawing
        CF_DISABLE_SPU_COLLISION_PROCESSING = 64//disable parallel/SPU processing
};
```

We observe that these flags are actually indicators of an allotted binary number. Each object as we have seen in our program previously are equipped not only with their own unique identifier but also this binary number which is used in combination with 'btRigidBody' functions and classes to help stipulate general collision detection demandments and protocols.

```
///to keep collision detection and dynamics separate we don't store a rigidbody pointer
        ///but a rigidbody is derived from btCollisionObject, so we can safely perform an upcast
        static const btRigidBody*       upcast(const btCollisionObject* colObj)
        {
                if (colObj->getInternalType()&btCollisionObject::CO_RIGID_BODY)
                        return (const btRigidBody*)colObj;
                return 0;
        }
        static btRigidBody*       upcast(btCollisionObject* colObj)
        {
                if (colObj->getInternalType()&btCollisionObject::CO_RIGID_BODY)
                        return (btRigidBody*)colObj;
                return 0;
        }
```

We can see that Bullet is capable of making distinctions between concepts such as general dynamic functions and collision detection response components, this allows our demonstration to proceed smoothly without potential errors or conflicts.

7.3 Collision Detection Algorithms

We present our examination of how the various classes of the Bullet library culminate to deliver a compelling collision detection system.

'btBroadphaseInterface' for providing bounding boxes:

```
///The btBroadphaseInterface class provides an interface to detect aabb-overlapping object pairs.
///Some implementations for this broadphase interface include btAxisSweep3, bt32BitAxisSweep3
///and btDbvtBroadphase. The actual overlapping pair management, storage, adding and removing
///of pairs is dealt by the btOverlappingPairCache class.
class btBroadphaseInterface
{
public:
        virtual ~btBroadphaseInterface() {}

        virtual btBroadphaseProxy*        createProxy(  const btVector3& aabbMin,  const btVector3&
        int shapeType,void* userPtr, short int collisionFilterGroup,short int collisionFilterMask,
        btDispatcher* dispatcher,void* multiSapProxy) =0;
        virtual void        destroyProxy(btBroadphaseProxy* proxy,btDispatcher* dispatcher)=0;
        virtual void        setAabb(btBroadphaseProxy* proxy,const btVector3& aabbMin,
        const btVector3& aabbMax, btDispatcher* dispatcher)=0;
        virtual void        getAabb(btBroadphaseProxy* proxy,btVector3& aabbMin, btVector3& aabbMax
        const =0;

        virtual void        rayTest(const btVector3& rayFrom,const btVector3& rayTo,
        btBroadphaseRayCallback& rayCallback, const btVector3& aabbMin=btVector3(0,0,0),
        const btVector3& aabbMax = btVector3(0,0,0)) = 0;

        virtual void        aabbTest(const btVector3& aabbMin, const btVector3& aabbMax,
        btBroadphaseAabbCallback& callback) = 0;

        ///calculateOverlappingPairs is optional: incremental algorithms (sweep and prune) might
        ///do it during the set aabb virtual void calculateOverlappingPairs(btDispatcher* dispatcher

        virtual        btOverlappingPairCache*        getOverlappingPairCache()=0;
        virtual        const btOverlappingPairCache*        getOverlappingPairCache() const =0;

        ///getAabb returns the axis aligned bounding box in the 'global' coordinate frame
        ///will add some transform later
        virtual void getBroadphaseAabb(btVector3& aabbMin,btVector3& aabbMax) const =0;

        ///reset broadphase internal structures, to ensure determinism/reproducability
        virtual void resetPool(btDispatcher* dispatcher) { (void) dispatcher; };

        virtual void        printStats() = 0;
};
```

This class is responsible for delegating priority assignments, and thus a queue of assignments that are to be processed accordingly. The assignment is to construct a axis-aligned bounding box that groups together close primitives/spheres in a clustered segment, this helps keep the collision detection algorithm persistent and without the possibility of error. 'btBroadphaseInterface' assists our demonstration when there is a large number of spheres populated in our scene and thus bounding box constructs are established which processes multiple but fast detection tests with the spheres contact points.

'btCollisionConfiguration' and 'btConstraintSolver' for ensuring consistent object behaviour:

```
///btCollisionConfiguration allows to configure Bullet collision detection
///stack allocator size, default collision algorithms and persistent manifold pool size
class        btCollisionConfiguration
{

public:

        virtual ~btCollisionConfiguration()
        {
        }

        ///memory pools
        virtual btPoolAllocator* getPersistentManifoldPool() = 0;

        virtual btPoolAllocator* getCollisionAlgorithmPool() = 0;


        virtual btCollisionAlgorithmCreateFunc* getCollisionAlgorithmCreateFunc
        (int proxyType0,int proxyType1) =0;

};


class  btConstraintSolver
{

public:

        virtual ~btConstraintSolver() {}

        virtual void prepareSolve (int /* numBodies */, int /* numManifolds */) {;}

        ///solve a group of constraints
        virtual btScalar solveGroup(btCollisionObject** bodies,int numBodies,
        btPersistentManifold** manifold,int numManifolds,btTypedConstraint**
        constraints,int numConstraints, const btContactSolverInfo& info,
        class btIDebugDraw* debugDrawer,btDispatcher* dispatcher) = 0;

        virtual void allSolved (const btContactSolverInfo& /* info */,class btIDebugDraw*
        /* debugDrawer */) {;}

        ///clear internal cached data and reset random seed
        virtual        void        reset() = 0;

        virtual btConstraintSolverType        getSolverType() const=0;


};
```

Both of these classes are Bullet's way of analysing the scene to ensure that the behavioural characteristics of all the spheres in our demonstration are kept consistent until termination. When it comes to our demonstration, there is not much to take into account since all objects are relatively the same and there is no other interactivity that needs to be considered other than with each other. These classes offer a way of governing the implications of what is happening with the scene in order to keep the general collision detection system consistent with implied consequences that occur since initiation of the program. Bullet ensures collision detection maintains its efficiency throughout our demonstration.

'btDispatcher' and 'btCollisionAlgorithm' to derive calculations for pairwise collision:

```
///The btDispatcher interface class can be used in combination with broadphase
///to dispatch calculations for overlapping pairs.
///For example for pairwise collision detection, calculating contact points stored
///in btPersistentManifold or user callbacks (game logic).
class btDispatcher
{


public:
        virtual ~btDispatcher() ;

        virtual btCollisionAlgorithm* findAlgorithm(const btCollisionObjectWrapper* body0Wrap,
        const btCollisionObjectWrapper* body1Wrap,btPersistentManifold* sharedManifold=0) = 0;

        virtual btPersistentManifold*        getNewManifold(const btCollisionObject* b0,
        const btCollisionObject* b1)=0;

        virtual void releaseManifold(btPersistentManifold* manifold)=0;

        virtual void clearManifold(btPersistentManifold* manifold)=0;

        virtual bool        needsCollision(const btCollisionObject* body0,
        const btCollisionObject* body1) = 0;

        virtual bool        needsResponse(const btCollisionObject* body0,
        const btCollisionObject* body1)=0;

        virtual void        dispatchAllCollisionPairs(btOverlappingPairCache* pairCache,
        const btDispatcherInfo& dispatchInfo,btDispatcher* dispatcher)  =0;

        virtual int getNumManifolds() const = 0;

        virtual btPersistentManifold* getManifoldByIndexInternal(int index) = 0;

        virtual        btPersistentManifold**        getInternalManifoldPointer() = 0;

        virtual        btPoolAllocator*        getInternalManifoldPool() = 0;

        virtual        const btPoolAllocator*        getInternalManifoldPool() const = 0;

        virtual        void* allocateCollisionAlgorithm(int size)  = 0;

        virtual        void freeCollisionAlgorithm(void* ptr) = 0;

};
```

When it comes to the matter of actual sphere-sphere collision then Bullet offers its own dynamic way of seeking correct the correct algorithms to apply. We notice that Bullet recognises our sphere 'rigid body' objects as of the type 'SPHERE_SHAPE_PROXYTYPE' hence this, in collaboration with 'manifold points' or contact points stipulates the desire for sphere-sphere pairwise collision. 'btDispatcher' typically takes this information and collaborates with 'btBroadphaseInterface' to establish the general sphere-sphere batch of algorithms that was discussed earlier. We can see then that Bullet is efficient in processing 'splitting' or fast paced testing within a bounding box until the final pairwise collision detections can be achieved.

```
///btCollisionAlgorithm is an collision interface that is compatible with the
///Broadphase and btDispatcher.
///It is persistent over frames
class btCollisionAlgorithm
{

protected:

        btDispatcher*        m_dispatcher;

protected:
//        int        getDispatcherId();

public:

        btCollisionAlgorithm() {};

        btCollisionAlgorithm(const btCollisionAlgorithmConstructionInfo& ci);

        virtual ~btCollisionAlgorithm() {};

        virtual void processCollision (const btCollisionObjectWrapper* body0Wrap,
        const btCollisionObjectWrapper* body1Wrap,const btDispatcherInfo& dispatchInfo,
        btManifoldResult* resultOut) = 0;

        virtual btScalar calculateTimeOfImpact(btCollisionObject* body0,btCollisionObject* body1,
        const btDispatcherInfo& dispatchInfo,btManifoldResult* resultOut) = 0;

        virtual        void        getAllContactManifolds
        (btManifoldArray&        manifoldArray) = 0;
};
```

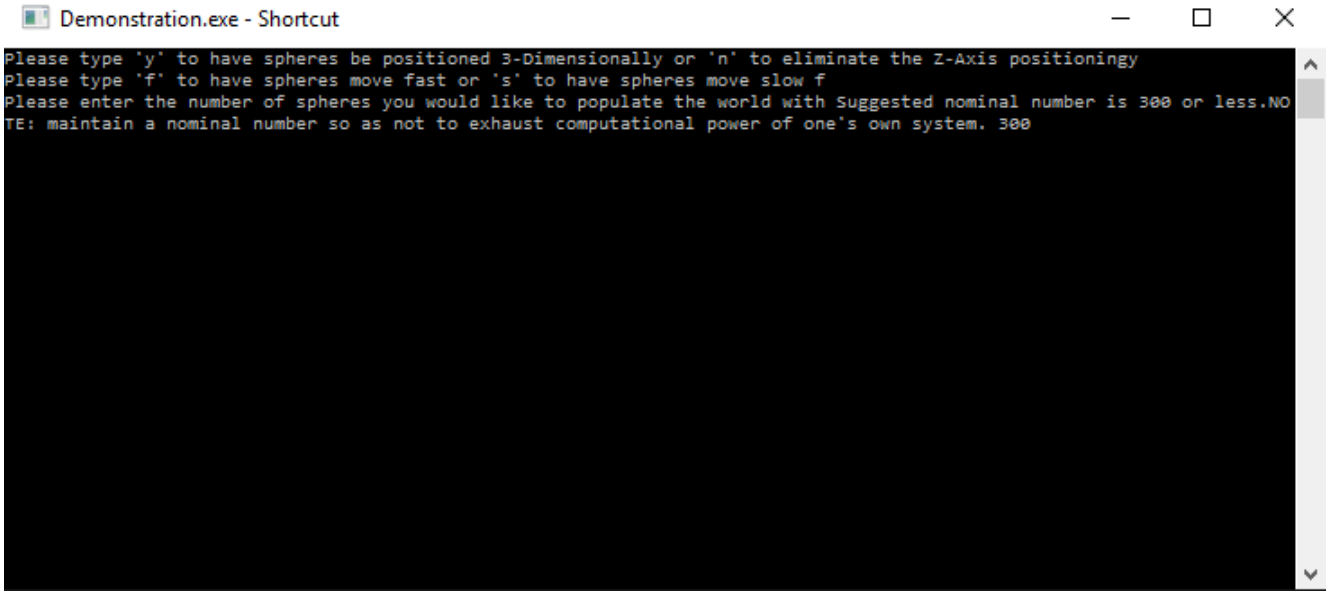
The class 'btCollisionAlgorithm' is a sub class that delivers the actual collision that was discussed. We notice this has a connection to the 'collisionDetection' method in our program by collecting each 'rigid body' as input and implementing the delivered collision algorithm using subsequent and derivative classes in the Bullet library.

8 Final Application

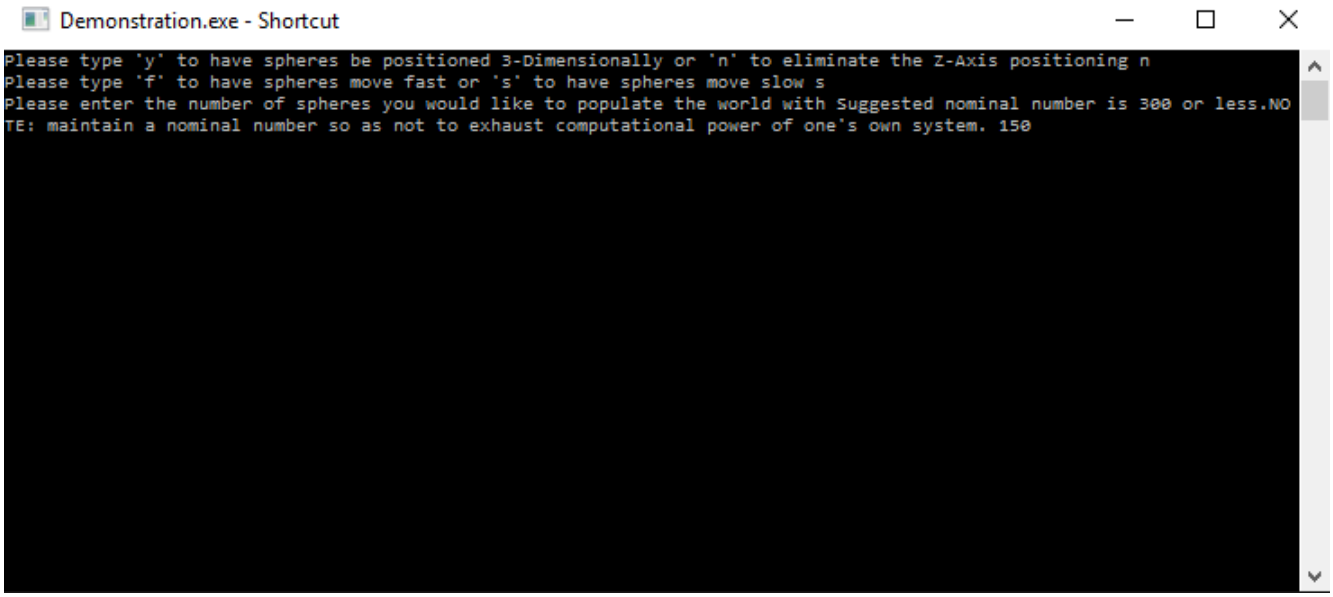
We present the final version of our demonstration.

Constructing the Scene

Scene preference settings are provided by the user.



The user can provide their own combination of these three settings.

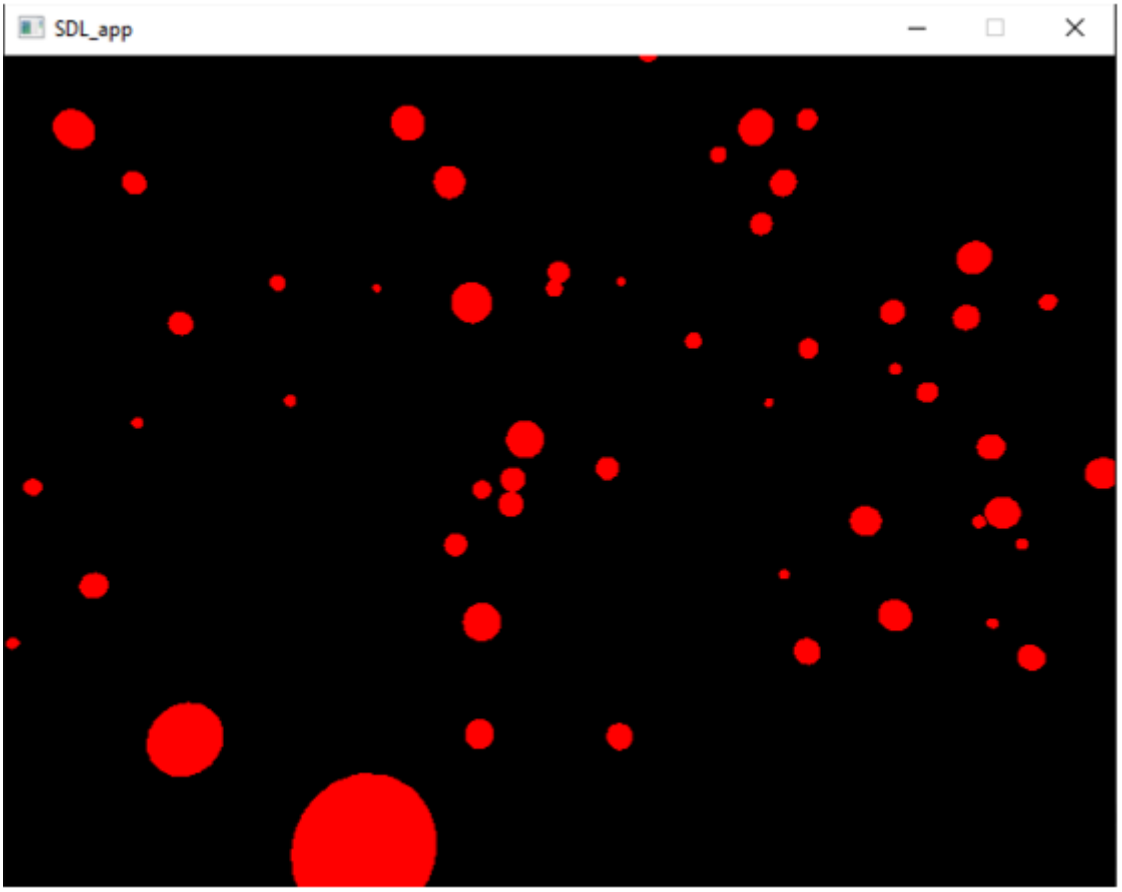


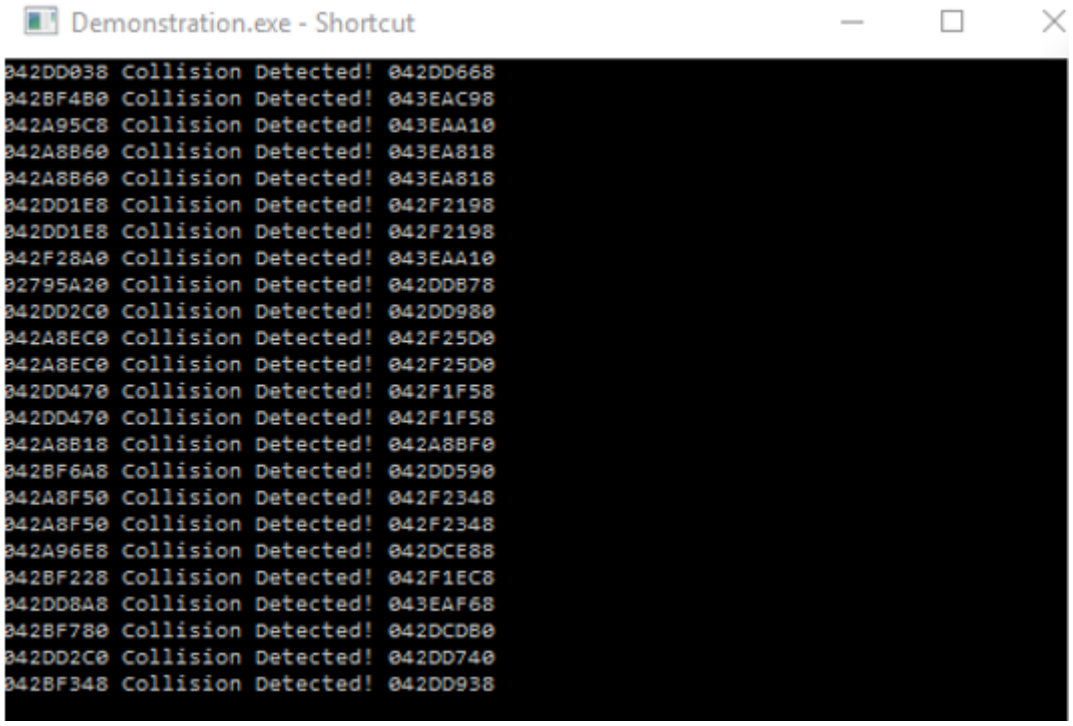
Thus the demonstration is customisable, the user may choose for the spheres to deploy only on the x and y axis to immediately see the effects of Bullet's organisation of clustered segments.

Collision Detection

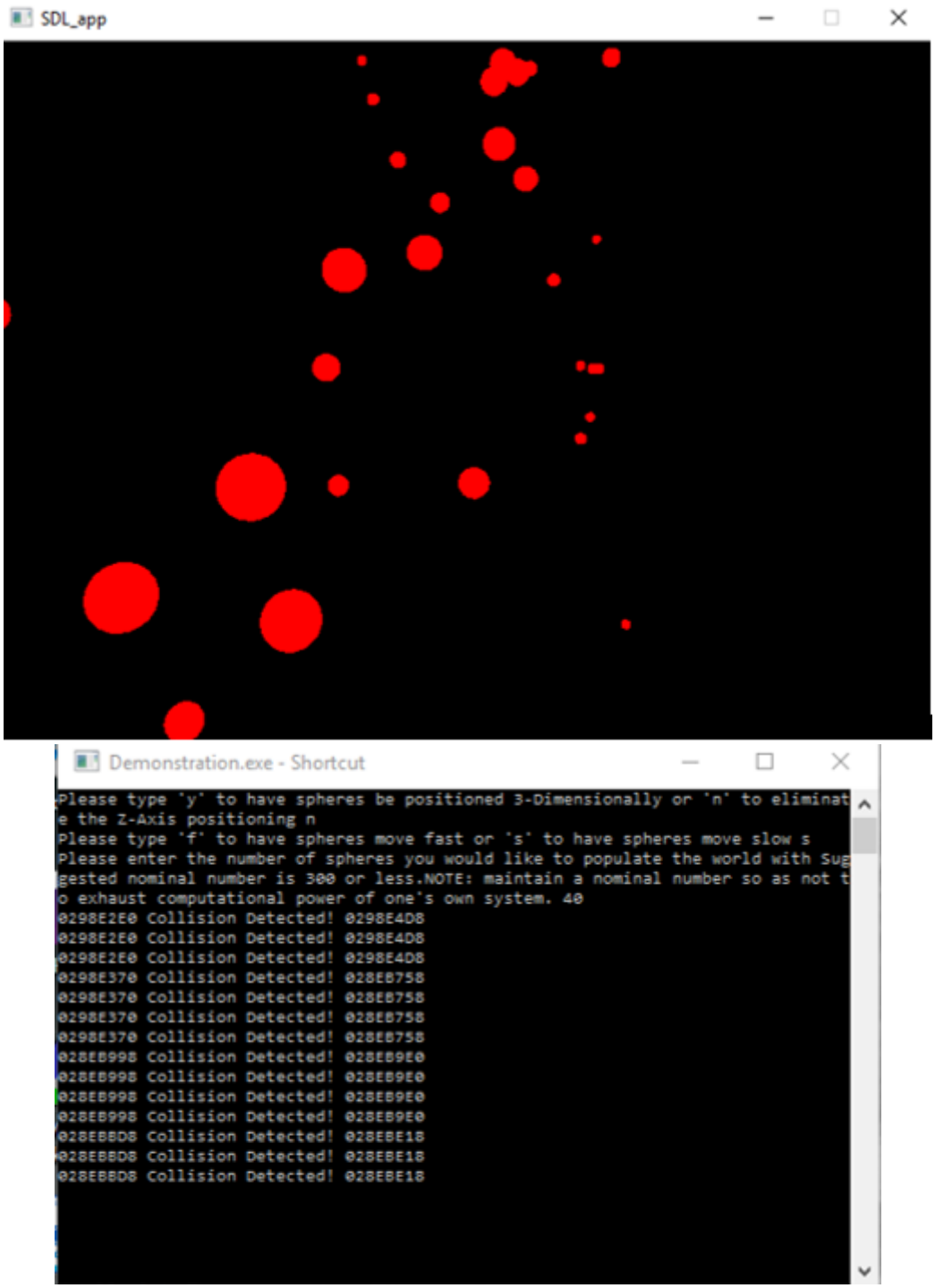
The following are instances of the use of the final application:

Our demonstration could be used to witness collision detection on a large scale.





Our demonstration could also be used to verify the effects of Bullet's collision detection by using a much more focused test scenario.



9 Analysis

The objective of our study is to document how the Bullet physics engine provides an elaborate deployment of collision detection. Firstly, this author insists that our demonstration is successful in conveying a successful object collision detection system. After several tests, this author can say with confidence that the system is entirely workable and operates in a coherent manner due to the reported output. Our demonstration allows us to examine Bullet and the various possibilities that it can provide. The demonstration proves to be valuable asset in exploring the Bullet library and offers a way to further experiment with more components and functions.

Unfortunately, due to time constraints this author was not able to configure our spheres to eventually return on their 'vector3' course so that the spheres could remain in relatively the same place or segment of the scene. However as stated the essential objective of this report has been achieved, and this can be further verified by the customisability of the demonstration accessible to the user. If the user wants to focus on observing collision detection then one way the user could do so would be to choose to have a slow paced environment to instill an opportunity for a more self contained study.

We see that Bullet is flexible in its approach to applying the critical mathematics to instill physics simulation. When using Bullet it was not necessary for us to program a concept such as pairwise collision ourselves for example but rather to simply provide the necessary information needed so that the Bullet library can be traversed so as to arrange and put together the scene and its behavioural characteristics that the user or programmer desired.

Additionally we observe that Bullet is very effective at implementing the commonly accepted general methods and concepts of collision detection that was discussed including establishing constructs such as bounding boxes throughout the scene to increase efficiency. What is more, we observe the collaboration between various functions designed to dynamically derive essential methodology and algorithms. There is a consistent structure of elegant and dynamic methods throughout the library that offer an opportunity to develop a real to life environment by correlating various concepts of physics to enable computer simulation.

When it comes to the discussion of how to better or more efficiently apply Bullet, one could argue that the library already handles detection through 'manifold points' and flag indicators designed specifically for collision callbacks very well, but of course we may learn a lot from our demonstration on how to develop even more elaborate ways of constructing this system in the future. We see from our examination of the library's functionality that there is a multiplicity of opportunities to purposely implement specific components in a deliberate manner. For instance, one could apply a certain strategy to our scene by utilising various apparatuses such as manual AABBs equipped with 'custom material callback' indicators for speedy and pre-calculated pairwise collision throughout the scene. We see that there are various way an expert user could apply Bullet in order to create a more elegant environment.

Finally, we take note that Bullet is substantial in its ability to construct elaborate environments and we assert that further use and practice with physics engines will yield greater opportunities.

10 Conclusion

In summary, this author suggests that our investigation has rewarded us with numerous answers for the questions we have on physics engines and their abilities.

Our demonstration is an effective way of analysing specifically the actions taken by physics engines and the role of such engines when creative software is being produced. Bullet, like many other physics simulation libraries/applications are evidence that the designer and programmer have a wide range of tools at their disposal.

We have learned a great deal of how the Bullet library assesses and further implements functionality to provide an advanced environment of which has virtually endless potential.

We have studied the Bullet libraries functionality and we dedicate this study as a guide to using Bullet in the future, so that we may further develop our skills with Bullet and improve our experience and skills with physics engines and how to successively accomplish efficient implementation for giving our applications a much more fluid and dynamic feel. Our investigation into this incredibly intricate topic will only continue with time, and we insist that part of our efforts into proceeding with our on-going study is engaging in more practice. Our demonstration consists of a greatly accessible invitation to object collision detection, and whether it be the application of rigid structures to create a comprehensive array of highly resourceful objects of interest or navigating through stored algorithms of great complexity, our exercise in our knowledge of this subject will only expand ever more.

Overall, our investigation with Erwin Couman's Bullet library has proved to be fruitful in our endeavour to ascertain and equip ourselves with a critical understanding of object collision detection as we make our foray into the inspiring world of physics simulation. There is only a great sense of anticipation as we continue our study and become a part of an ever increasing and evolving realm of expansive potential.

References

[Arv90] James Arvo. "A Simple Method for Box-Sphere Intersection Testing". In: *Graphics Gems*. Ed. by Andrew Glassner. San Diego, CA, USA: Academic Press Professional, Inc., 1990, pp. 335–339.

[BB07] Adrian Boeing and Thomas Braunl. "Evaluation of Real Time Physics Simulation Systems". In: *GRAPHITE '07 Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (2007), pp. 281–288. DOI: http://dx.doi.org/10.1145/1321261.1321312.

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection". In: *SIGGRAPH '96 Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 171–180. DOI: http://dx.doi.org/10.1145/237170.237244.

[HBC10] Hearn, Baker, and Carithers. *Computer Graphics with OpenGL*. 4th. 2010.

[Hub96] Philip M. Hubbard. "Approximating polyhedra with spheres for time-critical collision detection". In: *ACM Transactions on Graphics (TOG)* 15.3 (1996), pp. 179–210. DOI: http://dx.doi.org/10.1145/231731.231732.

[KGS97] Dong Jin Kim, Leonidas J Guibas, and Sung Yong Shin. "Fast Collision Detection among Multiple Moving Spheres". In: *SCG '97 Proceedings of the thirteenth annual symposium on Computational geometry* (1997), pp. 373–375. DOI: http://dx.doi.org/10.1109/CA.1997.601033.

[Klo+98] James T. Klosowski et al. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs". In: *IEEE Transactions on Visualization and Computer Graphics* 4.1 (1998), pp. 21–36. DOI: http://dx.doi.org/10.1109/2945.675649.

[LAL07] Thomas Larsson, Tomas Akenine-Moller, and Eric Lengyel. "On Faster Sphere-Box Overlap Testing". In: *Journal of Graphics, GPU, and Game Tools* 12.1 (2007), pp. 3–8. DOI: http://dx.doi.org/10.1080/2151237X.2007.10129232.

[Mol97] Tomas Moller. "A Fast Triangle-Triangle Intersection Test". In: *Journal of Graphics Tools* 2.2 (1997), pp. 25–30. DOI: http://dx.doi.org/10.1080/10867651.1997.10487472.

[MW88] Matthew Moore and Jane Wilhelms. "Collision Detection and Response for Computer Animation". In: *ACM SIGGRAPH Computer Graphics* 22.4 (1988), pp. 289–298. DOI: http://dx.doi.org/10.1145/378456.378528.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th. 2013.