Expertise
and insight
for the future

Erkki Parkkulainen

Real-time Physics Simulation

Metropolia University of Applied Sciences

Bachelor of Engineering

Information technology

Bachelor's Thesis

14 November 2019

Author Title Number of Pages Date	Erkki Parkkulainen 28 pages 14 November 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of School

The purpose of the study was the implementation of a 3D physics engine that is usable in real-time graphics applications such as games and other simulations where approximation of physics is adequate. Along with the actual physics engine an OpenGl based graphics toolkit was developed for testing purposes and visualization of the physics engine and for creating a demonstration application.

The thesis covers the fundamentals of physics simulation, integration of physics, collision detection and collision handling, as well as the abstraction of these concepts into a library that can be used in a separate application. The tools and practices used in the development of the graphics toolkit and the physics engine are also covered.

As a result of the study both the minimal graphics toolkit and the physic engine were created, and the project can be considered a success.

Keywords	Physics simulation, OpenGL, 3D

Tekijä Otsikko Sivumäärä Aika	Erkki Parkkulainen 28 pages 14.11.2019
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotuotanto
Ohjaaja	Janne Salonen, Tutkintovastaava

Työn tarkoitus on toteuttaa 3D-fysiikkamoottori, jota voi käyttää reaaliaikaisissa grafiikka-sovelluksissa kuten peleissä ja muissa simulaatioissa, joissa riittää approksimaatio fysiikasta. Fysiikkamoottorin lisäksi kehitetään OpenGL-pohjainen grafiikkakirjasto fysiikka-moottorin testausta ja visualisointia varten.

Työ kattaa fysiikkasimulaation perusteet, integroinnin, törmäysten havaitsemisen ja niiden käsittelyn. Nämä toteutetaan kirjastoiksi, joita voidaan käyttää muista sovelluksista. Lisäksi työ myös käsittelee työn tekemiseen käytetyt työkalut.

Työn tuloksena tuotettiin minimaalinen grafiikkakirjasto ja fysiikkamoottori, kuten oli suunniteltu.

Keywords	Fysiikka-simulaatio, OpenGL, 3D

Contents

1 Introduction

In 3D applications and 2D games it is often very easy to create interesting interactions, mechanics and realism just by adding a physics simulation. Even if the physics simulation is not strictly required it may provide a sense of realism and immersion. Games can be taken as an as an example, as games have developed from having no physics simulation to implementing a specific phenomenon on a case-by-case basis as required by game mechanics. The next step is to simulate physics effects that are not actually needed for the mechanics but are implemented only to achieve realism. This is analogous to the development of 3D rendering to achieve different mechanics and now to also approach photorealism. [1, p. 647]

This thesis focuses on real-time simulation and rendering of physics that is mostly useful for games and game like applications. Specifically, the purpose is to create a physics engine that can be used to create believable rigid body physics that can be used for several different types of simulations. The simplest ways to add a physics simulation to an application would be to use an existing physics engine or separately implement only the necessary functionality. The existing physics engines are of extremely high quality and full featured. The purpose of creating a new engine is to have the simplicity from the smaller feature set and still be extensible and general enough to be used in varied scenarios and to learn about the underlying concepts. [2] [3] [4]

These days many of the high-quality engines are open source and freely available without any license fees. One of the most common ones is PhysX, which is currently owned by Nvidia and was made open source in December 2018. However, for console platforms it is still proprietary, and some license costs may apply. PhysX is also the physics engine that is used both in Unity and Unreal game engines which are both hugely popular. One example of still a fully proprietary physics engine is Havok which was previously owned by Intel and used to be affordable and completely free for small scale games. After being purchased by Microsoft Havok SDK is at the time of writing no longer freely available but can probably be acquired by contacting their sales and negotiating a price. One additional example of a very high-quality open source 3D physics engine of the same segment is Bullet which has been open source since the beginning. For 2D physics, one example is Box2D which is also open source and is popular for indie development due to its simplicity of use. There are also dozens of other general-purpose and more specialized (ex, car physics, molecular physics, etc.) open source and commercial physics engines but for the most cases the ones mentioned above are more than good enough. [3] [2] [4] [5]

2 Physics Simulation

This chapter covers the basic components required for the real-time simulation of a physics engine.

2.1 Simulating Forces

At the core of the physics simulation is the component called integrator. When simulating physics, ultimately what the application needs from the physics engine is to know the position of the object being simulated after applying any forces. Applying forces to an object creates acceleration based on the properties of the object. The integrator receives the acceleration and calculates the integral which gives the velocity, which can be integrated again to figure out the change in position. For the purposes of a physics engine the integration of acceleration to velocity and velocity to position are the only ones required, so in the context of real-time physics simulation integration is synonymous to updating velocity and position. Angular velocity and orientation are updated with the same integration methods as the linear forces. [6, p. 43]

The integrator function is used to iterate through all the objects in the simulation and to update the current state based on the elapsed time. The application usually decides how often this is done but usual methods are to use a fixed time step for the simulation or to update the simulation once per frame. Semi-fixed strategies can also be used. For example, the simulation could be updated once per frame but there could also be a maximum time step in case there is an occasional slow frame that could make the simulation quirky. There are several integration methods commonly used and one common feature with these is that they approach the correct solution with the decrease of time step. This means that for realistic simulation the fixed time step is usually preferred because if the integrator is executed once per frame it is possible that the simulation becomes very inaccurate if the frame takes too long. This also means that there needs to be a balance between accuracy of simulation and performance, since if the time step is set to a too small a value then the simulation can no longer be run in real-time. [7]

Some common families of integration methods are Euler, Verlet and Runge-Kutta, other methods exist and there are also many different variations. The reason why multiple different integrators are needed is that no method is perfect for all cases and all have their strengths and drawbacks. The different characteristics of the different integration methods are performance, stability and accuracy. In addition, different integrator variants might have different stability characteristics based on the simulation type. One integrator might perform better with orbital motions and another might be more stable when handling constraints like stiff springs. One example of an integrator would be the RK4 which is a fourth order Runge-Kutta method that performs the evaluation four times as compared to Euler which is a first order method. This basically means that RK4 will be roughly four times slower than Euler, but it also converges faster giving more accurate and stable results compared even when running the Euler methods with smaller time-step gaining back some time that is lost from the more complex calculations. The Verlet integration method is a second order method and provides better stability than Euler and is only slightly slower. [8] [9] [10]

The choice of the integration method depends on the application needs. The Euler method is the fastest of the ones discussed here, but also the most inaccurate and unstable, RK4 is the most accurate but slowest and Verlet would offer balance between the two. For games the accuracy of Euler would usually be enough, unless there are some specific cases like stiff springs or constraints. RK4 should probably be reserved for special cases where the added precision is required for the simulation, but it would be difficult to argue against Verlet. One added consideration is that the accuracy and stability also depend on the type of simulation, for example Euler would be more stable with constant velocities. Formula 1 describes one variant of the Euler method. [6, pp. 417-418] [10] [11]

$$v_{n+1} = v_n + a_n \Delta t \tag{1}$$
$$x_{n+1} = x_n + v_{n+1} \Delta t$$

v is the velocity

a is the acceleration

x is the position

Δt is the timestep.

Formula 1. Euler-Cromer integration method

Formula 1 describes the Euler-Cromer integration method, also called the semi-implicit Euler-method and it is a modification to the base Euler-method. First, the velocity is approximated, and then the same method is used to approximate the position based on the approximated velocity. In the base Euler-method the original velocity would instead be used in in the position update too, but this would produce more unstable results since the second order term is not used. The reason for this is that applying the second order term $\frac{1}{2} a_n \Delta t^2$ in the position update would most often not make a significant difference in high frame rate applications since the time step is very small and would only make the calculations more intensive. The semi-implicit Euler-method is more stable than the base Euler, but it is still a first order method. If the application has large enough acceleration values to make the second order term significant or if just higher accuracy is necessary, then using the second order version of the Euler integration instead can be beneficial. The version of Euler with the second order term included is also called modified Euler method or the midpoint method. An example of a different approach is given in Formula 2. [6, pp. 417-418] [12]

$$x_{n+1} = 2x_n - x_{n-1} + a_n \Delta t^2 \tag{2}$$

a is the acceleration

x is the position

Δt is the timestep.

Formula 2. Verlet integration method.

Formula 2 describes the Verlet integration method which unlike the Euler integration does not store the velocity-state at all and instead uses the previous position in the formula to account for it. For a single-body simulation Verlet will yield similar accuracy as the Euler but it will preserve energies more accurately when dealing with constraints. Accuracy of the position for this method is third order but it does not account for the velocity. The velocity can be calculated simply by using the positions, but the accuracy will be of the first order and since it is a derived property, it cannot be modified externally. Formula 3 depicts a more useful variant of the Verlet method. [13]

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2 \qquad (3)$$

$$v_{n+1} = v_n + \frac{a_n + a_{n+1}}{2} \Delta t$$

v is the velocity

a is the acceleration

x is the position

Δt is the timestep.

Formula 3. Velocity verlet integration method.

Formula 3 shows a variant of Verlet integration called velocity Verlet. Velocity Verlet is for the present purposes a more convenient version of the Verlet integration since it explicitly includes the velocity which can be very useful for example when resolving collisions. In contrast to the base Verlet method the velocity Verlet is second order for both the position and the velocity. [13]

2.2 Collision Detection

With the integrator function realistic movement can be simulated but if the interaction between different objects is required then this is not enough. For this effect, collision detection is required. Collision detection is the process of figuring out whether objects are in contact and generating the contact information for the contact points. This is a very broad subject and only the basics are covered here.

If only primitive shapes are used, then performing the intersect tests is reasonably simple. Testing whether two spheres are penetrating is trivial, but more complex shapes become more computationally heavy. For more complex shapes two common generic algorithms are the separating axis theorem (SAT) and Gilbert-Johnson-Keerthi (GJK). The SAT algorithm works by creating a projection of the shapes for an axis and checking if the projections overlap, if they do not, the shapes are not intersecting. The test can then be repeated for all relevant axes based on the edges and faces of the two shapes. If any of these tests fail, then the objects are not in contact. In general, the SAT should perform well for 2D shapes and for primitive 3D shapes but the number of axes that needs to be tested grows quadratically with the complexity of the shape. GJK would generally be more efficient for 3D environments, but unlike SAT which would already provide the data for the contact information the GJK only finds out if the objects are in contact and their positions. This means that for physics simulation it usually needs to be supplemented with some other method to generate the full contact information that is needed. One example is Expanding Polytope Algorithm (EPA), which relies on the same principles as the GJK. [6, p. 319] [14, p. 399] [15] [16] [17]

One thing that both the GJK and SAT methods have in common is that they require the shapes being tested to be convex and additionally for SAT the shapes need to be polyhedrons. For a shape to be convex a line between any two arbitrary points inside the shape must be contained within the shape. For a concave shape a line between these points might intersect the borders of the shape. This means that if collision between concave shapes is necessary some extra work or a more complex algorithm is required. Instead of using a more complex collision algorithm, it is more common to either create convex hull for the concave shape or use decomposition. Convex hull means that a new minimal convex shape is generated that contains the concave sacrificing the inward folds in the shape, which might be acceptable depending on the object and the use case. Another method is to use decomposition and split the model to smaller parts where each of these are convex. This can be done beforehand when creating the collision mesh or it can be done programmatically, but for performance it would be best if the collision meshes are concave to begin with. [18] [14, p. 500]

Since the collision detection is a very heavy operation, it is usually unfeasible to test all the objects in the scene for collisions. To solve this problem the collision detection can be split to broad-phase and narrow-phase. The point of broad-phase is to make a rough estimation of which objects might be in contact and generate smaller sets that can be then used in the narrow-phase to find the actual contacts. One way to achieve this would be to construct a bounding volume hierarchy (BVH) where a tree structure of bounding volumes is constructed, where the bounding volumes contain objects that are near each other. The hierarchy is implemented as a tree that can contain several levels of bounding volumes and can be traversed with collision detection being performed on the bounding volumes on each level until there is a set of objects left that are likely to have collisions. This set can then be passed to the narrow phase to perform the full collision detection. Alternative strategy would be spatial partitioning. In spatial partitioning the space would be split into regions. Then objects within each region can be checked for collisions and if an object intersects a boundary of regions it is possible it might collide also with objects from the adjacent region. [6, p. 255] [14, p. 235] [14, p. 285] [19] [20]

2.3 Resolving Collisions

The collision subsystem produces a list of all collisions. Each collision data structure contains the information for the contact point, contact normal and the penetration depth. This information is then passed to the collision resolver along with the rigid bodies. The rigid bodies contain the data from the physics simulation including velocities, positions, orientation and other properties which can be used along with the contact information to figure out what should be the result of the collision.

When two objects collide in real world what happens is that these objects will compress and deform. How much they deform depends on the material and the compression is often imperceptible and it is this type of object that does not visibly deform that is represented by the rigid body physics. After the deformation depending on the material there is a force for the object to return to its original shape. This defines how much bounce the object will have. In the physics engine this property can be represented with the coefficient of restitution which is the ratio of the velocity after the contact and the initial velocity and will provide the change in velocity from the contact. [6, p. 115] [21]

Due to the time step not being infinitely small the collisions often result in interpenetration of the objects. This does not happen in real life and these issues must first be resolved. To resolve these, one strategy would be to move both objects along the contact normal. The mass of the colliding objects also must be considered when resolving the collision. If there was a collision between a feather and a bowling ball it would hardly make sense if both were moved by the same amount nor would it appear natural. Intuitively it would be more natural to mostly move the feather and even if the bowling ball was affected it should be by a very minute amount. This can also be seen in Formula 4 for resolving linear interpenetration.

$$\Delta x_a = \frac{m_b}{m_a + m_b} * pn \qquad (4)$$

Δx_a is the position change of first object.

m_a is the mass of the first object.

m_a is the mass of the second object.

p is the penetration depth.

n is contact normal.

Formula 4. Updating position by mass ratio.

As can be seen in Formula 4, the positions of the objects are updated based on the inverse proportion of their masses. This will have the effect that if the objects have equal mass, they will be moved by the same amount and in the case that the collision is between an object with tiny mass and object the result will appear more intuitive. [6, p. 124]

When resolving collisions, the velocity can be updated directly, and there is no time step provided. The collision resolution can however directly resolve the collisions as velocity changes by using concept of impulse forces. Impulse is a force that can be applied immediately. Below some descriptions of impulse and its relation to force (Formula 5).

$$i = ft \qquad (5)$$

$$f = ma$$

$$i = mat$$

$$at = \Delta v$$

$$i = m\Delta v$$

i is the impulse.

f is the force.

m is the mass.

a is the acceleration.

Δv is the velocity change

Formula 5. Definition of impulse.

In Formula 5 if in the first equation for impulse the force is expanded as ma, the at can then be replaced with the delta velocity. Based on this information the collision resolving can work with impulses, first the impulse is generated on contact which can then be directly resolved as the velocity change based on mass of the object. Below the formula for generating the impulse (Formula 6). [22]

$$i = \frac{v_d}{v_i} \tag{6}$$

i is the impulse.

v_d is the desired velocity change.

v_i is the velocity change per unit impulse.

Formula 6. Calculating impulse.

Formula 6 is for generating the impulse based on the information that is provided by the collision detection step. The desired velocity change can be computed based on the coefficient of restitution and the contact velocity. The velocity change per unit impulse can be computed from the contact position, contact normal and mass of the object. The formulas above however only consider the linear component of the collision resolution. To realistically resolve the collisions the inertia, rotation and angular velocities would also have to be considered. They work on similar principles as the linear components featured here, but accounting for them makes things more complicated since they need to be represented with vectors in three-dimensional space. In order to keep the explanations simpler, the angular components have been omitted from this introduction to the subject. One example of the events seen when accounting for the angular components would be that if a pen is dropped in a tilted position so that one side of it hits the ground first, what happens is that only the contact position bounces but the center mass of the pen will still keep falling downwards. Without accounting for the angular components, the pen would bounce upwards without a change in its orientation.

3 Physics Engine Specification

This chapter covers the different subsections of the implementation of the physics engine and what was implemented.

3.1 Rendering Framework

The minimal rendering framework provides an abstraction of OpenGL and implements an object-class that represents objects inside the world. This class encapsulates both the rendering functions and functions for updating the physics state. On the high-level the main loop calls the update function on a fixed time step for each object and then similarly calls the render-function to display the current state of objects in the world.

3.2 Physics Engine Features

The physics engine implements support for rigid body physics. Only fundamental features of rigid body physics are not implemented. Other types of simulations such as soft body physics and fluid physics are out of the scope of the implementation. In addition, some simpler features such as constraints are left out for now.

3.3 Rigid Bodies

The rigid body represents the current state of a single object in the world and the objects properties that relate to the physics simulation. It also implements the high-level functionality for updating the state based on current forces and functions for adding new forces.

3.4 Collisions

Collision detection is left out of the scope of the implementation and instead external library is used, since implementing full featured collision detection would be too large of a task. Implementing a simple collision detection that only supports primitive shapes would still be doable but would increase the scope of the project and would not be as flexible as the separate library.

The physics engine does, however, implement the collision resolver that goes through the collisions detected by external library and implements the algorithm that applies the velocity and position changes caused by the collisions. The abstraction for collisions is implemented and constructed based on the information provided by the external library and implements the functions for calculating the position and velocity changes based on the collisions.

4 Physics Engine Implementation

For the implementation C++-language was used due to its popularity in high performance applications.

4.1 Project Management

This subsection covers the tools and practices used to manage the project development.

4.1.1 Version Control

Version control is a very important aspect of managing software development. It is especially important in large projects where different revisions written by multiple people need to be managed and will facilitate separate development efforts, allow rolling back to earlier revisions when necessary and make it easier to handle conflicts between different changesets. It is also extremely helpful even in projects managed by only one person. For this project Git was used as it is currently effectively the de facto standard that many companies are using or are migrating to. There are also several reputable free cloud services for hosting the git-repositories, like GitHub, Gitlab and Bitbucket. [23] [24] [25] [26]

4.1.2 Build System

The project build was managed with CMake which is the current de facto build system for C++ projects. Some traditional build systems such as Autotools and MSBuild are still widely used, but most new and actively maintained projects are moving to CMake. Other promising build systems which offer clearer syntax and some improved features are also emerging, for example Meson which some notable open-source projects have already adopted, but they have yet to supersede CMake as the de facto build system. [27] [28]

4.1.3 Project Structure

There are some guidelines for project structure, but there is no single standard used and the split is largely matter of taste and is guided by type of the project and tools used. For this project the following split was used:

- “cmake”, for build-system modules and scripts.
- “examples”, for test applications and examples.
- “src”, for source code.
- “include”, for headers of the public api.

Mixing the source and header files is also common with the advantage that it makes it easier to switch between the header and implementation. However, most code editors these days are able to swap between them even if they are stored in different directories. The benefit of having the headers in separate folder is cleaner separation of the public and private application programming interface and simpler packaging since the include directory can be copied as is. If the project would consist of several sub-libraries and executables different project structure would be required and there would be several reasonable ways of splitting the project, but for a single library project the simple split featured above is sufficient. [29]

4.2 Rendering

For visualization and testing of the physics engine a minimal framework is written from scratch along with a simple scene viewer that was used as test application for integration of the physics engine. OpenGL was used for the rendering and Epoxy-library will be used for OpenGL function pointer management. GLFW-framework was used for window management and input handling. GLM math library was used for the matrix calculations needed for the scene viewer implementation. Assimp-library was used for loading 3D mesh data. [30] [31] [32]

To ease the use of OpenGL some abstractions were created that wrap the raw OpenGL usage. Texture-class for binding image data for drawing, shader-class for wrapping shader-programs and mesh-class for representing object shapes. On top of these, some high-level abstractions were implemented. The high-level abstractions that were added are material-class for representing properties of the object surface such as texture and diffuse that are passed to the shader-program, model-class for combining mesh-data to a material and object-class for composing different components e.g. the 3D graphics models and physics engine rigid bodies. In addition, the resource manager class was implemented for loading image data, 3D mesh data and shader programs. Camera-class was also implemented for managing the view of the scene. Listing 1 contains a minimal example of the top-level of the scene viewer application.

```
while (true)
{
      current = std::chrono::steady_clock::now();
      accumulator += std::chrono::duration_cast<Microseconds>
            (current - previous);
      previous = current;

      while (accumulator >= dt)
      {
            update(dt)

            accumulator -= dt;
      }

      render()
}
```

Listing 1. The main loop.

In Listing 1 there is an example of the main loop with physics simulation. An alternative would be to directly use the measured frame time, but that can result in the simulation being affected by the stability of the rendering performance and performance of the hardware being used. In the implementation described here the constant dt can be a constant defined based on the application needs and the physics simulation will always be updated by this constant time step. Using too small a time step here might cause the physics calculations to dominate the running time and will lower the frame rate, too large a time step on the other hand might make the simulation unstable. A rough example of a proper time step for the type of applications this project is aimed at would be in the range of 10-30 milliseconds. Listing 2 depicts setting up the scene using the developed graphics toolkit.

```
ResourceManager resourceManager;

Camera camera(glm::vec3(0.0f, 3.0f, 7.0f), glm::vec3(0.0f, 1.0f, 0.0f),
      glm::vec3(0.0f, 0.0f, -1.0f));

glm::mat4 view = camera.getViewMatrix();
glm::mat4 projection = glm::perspective(45.0f, (float)screenWidth /
      (float)screenHeight, 0.1f, 100.0f);

Shader* shader = resourceManager.loadShader("minimal.vert", "minimal.frag");
shader->bind();
shader->setUniform("projection", 1, projection);
shader->setUniform("view", 1, view);

Model* model = resourceManager.loadModel("cube.obj", shader);

RigidObject obj("cube");
obj.addModel(model));
```

Listing 2. Setting up a scene.

As shown in Listing 2, first, the resource manager instance is constructed which is responsible of both implementing code for loading resources and managing the lifetime of them. Then the camera object is created which takes as parameters the position of the camera in 3D space, the up vector and the direction the camera is looking at. The projection matrix is also calculated and after the shader has been loaded using the resource manager both the view matrix and projection are set as uniform variables to the shader. The model loading takes as parameters the path to the model file and the shader which is needed because the model loading also constructs the materials based on the model file and the shader program is required for constructing the materials. Finally, everything is pulled together by constructing the rigid object which is the physics enabled base object type in the scene viewer demonstration application.

4.3 Physics

For the vector, the matrix and quaternion calculations GLM math library was used. For collision detection the FCL library was used. [33] [34]

4.3.1 Rigid Bodies

Rigidbody class is the main implementation for representing objects in the rigid body physics simulation. It contains the state representation of the object e.g. position, velocity and acceleration. The most important function it implements is the integrate function which calculates the new acceleration and angular acceleration based on the accumulated forces and properties of the object such as mass and inertia. Then according to the new acceleration, the new velocity, position and orientation of the object is integrated.

The class also implements functions for applying forces. There are separate functions for adding linear force which can be used for constant forces like gravity and for adding a force at a point which will also add angular torque in addition to the linear force based on the contact position. Damping is also added to decay the forces to roughly approximate things like air resistance. Listing 3 demonstrates the core functionality of this class.

```
void Rigidbody::integrate(float duration)
{
     previousAcceleration = acceleration;

     previousAcceleration += inverseMass * forceAccum;

     glm::vec3 angularAcceleration = torqueAccum * inverseWorldInertiaTensor;

     velocity += duration * previousAcceleration;
     rotation += duration * angularAcceleration;

     velocity *= std::pow(linearDamping, duration);
     rotation *= std::pow(angularDamping, duration);

     position += duration * velocity;

     glm::quat q(0.0f, rotation.x, rotation.y, rotation.z);
     orientation += q * orientation * duration * 0.5f;
     orientation = glm::normalize(orientation);

     transform = glm::mat4_cast(orientation);
     glm::mat4 translation = glm::translate(glm::mat4(), position);
     transform = translation * transform;

     glm::mat3 rotation = glm::mat3(transform);
     inverseWorldInertiaTensor =
          rotation * inverseInertiaTensor * glm::transpose(rotation);

     forceAccum = glm::vec3(0.0f);
     torqueAccum = glm::vec3(0.0f);
}
```

Listing 3. Implementation of semi implicit Euler integration.

Listing 3 provides the implementation of the integrate method. First, the accumulated forces and torque are converted to linear and angular acceleration based on properties of the object. Then the angular and linear velocities are updated based on the length of the time step provided as the argument to method. A similar step is repeated by using newly approximated velocity to calculate the new position and orientation of the object. Both steps increase in accuracy as the value of the time step is reduced, and with too large of a time step the simulation will become increasingly inaccurate. In between the linear and angular velocities are also decayed by the damping factors. At the end the accumulated forces are cleared since they have already been accounted for. Aside from the integrate-method, most of the rigid body implementation is quite simple. It mostly consists of just setter and getter methods for the different properties of the rigid body. Below in Listing 4 there is an example of one useful method in the rigid body class besides the integrate method.

```
void Rigidbody::addForceAtPoint(const glm::vec3& force,
	const glm::vec3& point)
{
	glm::vec3 pt = point;
	pt -= position;

	forceAccum += force;
	torqueAccum += glm::cross(pt, force);
}
```

Listing 4. Adding force with angular component.

Listing 4 introduces a helper method that in addition to the force, also takes the point where the force is applied from. With this information the torque-can be calculated. One example for a use-case for this method would be hitting a billiard ball on the side which introduces a spin to the ball in addition to the linear forces.

4.3.2 Collisions

The collision implementation consists of two separate classes; a collision for representing separate collisions and a collision resolver that implements higher level algorithm for resolving all collisions.

Starting from top down the collision resolver has a couple of parameters such as maximum iterations and epsilon. The algorithm loops through all collisions passed to it and call the collision-class to adjust the positions and velocities. This is done iteratively, first adjusting positions to fix interpenetration between the collision objects starting with the collision with the worst penetration. This is repeated until the maximum iteration count is reached or the highest penetration is smaller than the epsilon value. Afterwards similar process is repeated for velocity changes starting with the collisions that will result in largest velocity change first. The importance of the iterative process is that if one interpenetration issue is fixed, it might cause the interpenetration at other collision points to get worse. The reason for resolving the high magnitude collisions first is because those are the most likely collision to affect the other collision points.

By far the most complex part of the physics engine implementation covered here is the math for handling the position and velocity change of the objects. The reason this is complicated is that not only is there a need to figure out the change in velocity, the inertia and torque must also be considered. For example, if a rotating object is dropped to the ground it will bounce in a different manner than a non-rotating object. Related to this, objects with different inertia will also behave differently on contact. Friction of the contact surface will also affect the result. To perform the position change, first the inertia of both bodies is calculated and then based on the inertia the linear and angular change necessary to resolve the penetration in a manner that appears natural is performed. For velocity change an impulse is generated that represents the velocity change due to rotation and linear motion. When generating the impulse, the friction must also be accounted for since it might change the direction of the impulse. The impulse is then applied as velocity and torque to the object. Listing 5 below has an example of calculating the impulse.

```
glm::vec3 Collision::calculateFrictionlessImpulse(
	const std::array<glm::mat3, 2>& inverseInertiaTensors)
{
	float deltaVelocity = 0.0f;

	for (size_t i = 0; i < bodies.size(); ++i)
	{
		if (bodies[i])
		{
			glm::vec3 deltaVelocityWorld =
				glm::cross(relativeContactPosition[i], contactNormal);
			deltaVelocityWorld =
				inverseInertiaTensors[i] * deltaVelocityWorld;
			deltaVelocityWorld =
				glm::cross(deltaVelocityWorld,
					relativeContactPosition[i]);

			deltaVelocity += glm::dot(deltaVelocityWorld, contactNormal);

			deltaVelocity += bodies[i]->getInverseMass();
		}
	}

	return glm::vec3(desiredDeltaVelocity / deltaVelocity, 0, 0);
}
```

Listing 5. Calculating impulse without friction.

In Listing 5 the code first calculates the delta velocity which is the velocity change per unit of impulse. In the calculations, both the mass and inertia of the objects are taken into account. The desired delta velocity has been calculated beforehand using the coefficient of restitution. The resulting impulse is a vector, but here only one element of it has a value. The reason is that the impulse calculated in this example is frictionless and if instead friction was taken into account the resulting impulse could be in a different direction. Below in Listing 6 an example of applying the impulse.

```
void Collision::applyVelocityChange(glm::vec3 velocity[2],
	glm::vec3 rotation[2])
{
	std::array<glm::mat3, 2> inverseInertiaTensors;
	inverseInertiaTensors[0] = bodies[0]->getInverseWorldInertiaTensor();
	if (bodies[1])
	{
		inverseInertiaTensors[1] =
			bodies[1]->getInverseWorldInertiaTensor();
	}

	glm::vec3 impulseContact;
	impulseContact = calculateFrictionlessImpulse(inverseInertiaTensors);

	glm::vec3 impulse = contactToWorld * impulseContact;

	std::array<glm::vec3, 2> impulseTorques;
	impulseTorques[0] = glm::cross(relativeContactPosition[0], impulse);
	impulseTorques[1] = glm::cross(impulse, relativeContactPosition[1]);

	for (size_t i = 0; i < bodies.size(); ++i)
	{
		if (bodies[i])
		{
			float sign = i ? -1.0f : 1.0f;
			rotation[i] = inverseInertiaTensors[i] * impulseTorques[i];
			velocity[i] = glm::vec3(0.0f);
			velocity[i] += sign * impulse * bodies[i]->getInverseMass();

			bodies[i]->addVelocity(velocity[i]);
			bodies[i]->addRotation(rotation[i]);
		}
	}
}
```

Listing 6. Applying of velocity change.

In Listing 6 the impulse generated in Listing 4 is converted to world space. Then, based on the impulse, the torques are calculated which based on the inertia of the objects add angular velocity. Linear velocity is a little simpler and can be calculated directly with the mass. One tricky detail in this implementation is that the change of velocities is propagated back to the caller through the parameters of the method. This is done because after changing the velocities all the collision points on the affected rigid bodies must be updated to account for the changes made. Listing 7 shows the update to the collisions.

```
primary->applyVelocityChange(velocityChange, rotationChange);

for (auto&& secondary : collisions)
{
      auto&& primaryBodies = primary->bodies;
      auto&& secondaryBodies = secondary->bodies;

      for (size_t primaryIndex = 0;
            primaryIndex < primaryBodies.size();
            ++primaryIndex)
      {
            for (size_t secondaryIndex = 0;
                  secondaryIndex < secondaryBodies.size();
                  ++secondaryIndex)
            {
                  auto&& primaryBody = primaryBodies[primaryIndex];
                  auto&& secondaryBody = secondaryBodies[secondaryIndex];
                  if (primaryBody == secondaryBody)
                  {
                        glm::vec3 deltaVel =
                              velocityChange[secondaryIndex] +
                              glm::cross(
                                    rotationChange[secondaryIndex],
                                    secondary->relativeContactPosition[
                                          primaryIndex]);

                        secondary->contactVelocity +=
                              (glm::transpose(secondary->contactToWorld) *
                              deltaVel) * (primaryIndex ? -1.0f : 1.0f);
                        secondary->calculateDesiredDeltaVelocity(duration);
                  }
            }
      }
}
```

Listing 7.   Applying velocity change to collision and updating the collisions.

In Listing 7 there is the call to the function from Listing 5 and the updating of the collisions. Primary here refers to the collision currently being resolved and after updating the velocity for it the all the current collisions are looped through as the secondary. Next, if any of the bodies in the secondary match the one in primary it means the collision that was just resolved will affect the collision found by the comparison. To fix this, the contact velocity from the collision object is adjusted to match the current state and the internals are updated. The selection of the primary collision is done by the following code (Listing 8):

```
Collision* primary = *std::max_element(collisions.begin(), collisions.end(),
      [](Collision* first, Collision* second) -> bool {
            return first->desiredDeltaVelocity < second->desiredDeltaVelocity;
      });

if (primary->desiredDeltaVelocity < m_velocityEpsilon)
{
      return true;
}
```

Listing 8. Selection of which collision to resolve.

In Listing 8 the element with the highest desired delta velocity is selected and then if the value is lower than the epsilon, the function will return true to signal the collision resolving that there is no longer need to continue resolving collisions since all the collisions have been resolved to satisfactory degree. Finally, below in Listing 9 there is the top-level algorithm of the collision resolving.

```
if (!collisions.empty())
{
      for (auto&& collision : collisions)
      {
            collision->calculateInternals(duration);
      }

      for (int i = 0; i < m_positionIterations; ++i)
      {
            if (adjustPositions(collisions))
            {
                  break;
            }
      }

      for (int i = 0; i < m_velocityIterations; ++i)
      {
            if (adjustVelocities(collisions, duration))
            {
                  break;
            }
      }
}
```

Listing 9. Collision resolving algorithm.

The algorithm in Listing 9 first calculates the internal values derived from the contact information needed when calculating the actions that have to be taken to resolve the collisions. Then the positions are iterated until either the maximum iteration limit is reached, or the adjustPositions function returns true to signal that the worst penetration depth is below the epsilon value. The same is repeated for the velocity. The iteration count is needed since if all the collisions were resolved completely, the runtime could end up being very large and as long as the worst collisions have been resolved the result should be acceptable. However, if the iteration count is too small some weird effects e.g. objects slowly falling through the floor might occur. The listings above covered only the velocity update in-depth but the position update works in the same manner, only the math inside the functions is different.

5 Results

The physics engine and the minimal rendering framework were successfully implemented within the planned scope. Except for collision detection, the core math and algorithms were implemented. The current implementation would be perfectly usable with some limitations but could still be improved in many ways.

One issue is that the application programming interface is not yet refined, some information that is needed by the implementation is currently passed in the user-side API, but by some design changes these could be hidden in the implementation making the use of the engine easier. Additionally, some helper classes to manage to world state could be added for easier integration of the engine to projects. Currently these are handled in the scene viewer implementation, but they should be generalized and moved to the physics engine. Additionally, the implementation of the collision resolving is not very refined.

Another area of improvement would be stability. The current implementation has some inherent problems that need some additional considerations to mitigate. For example, in cases where there is continuous contact between multiple objects at the same time since the collisions are resolved one by one and this can result in some surprising effects if the resolution of one collision affects other collisions. The stability issues could however be worked on case-by-case manner if they become issues when using the engine for some specific application.

One more area of improvement would be performance. If the engine would be used in truly large-scale project, the current implementation would be naive. However, at the current state the performance should already be sufficient for any small scale and demo-projects that the engine would feasible be used for. Some ways to improve performance would be to improve the locality of data, so that it can be more efficiently cached and to add vectorization to perform several calculations simultaneously. This would however make the implementation unnecessarily complex and remove the advantage of simplicity and ease of modification, and in this case, it would be better to just use some existing solution. Figures 1, 2, 3 and 4 demonstrate the results of the project.

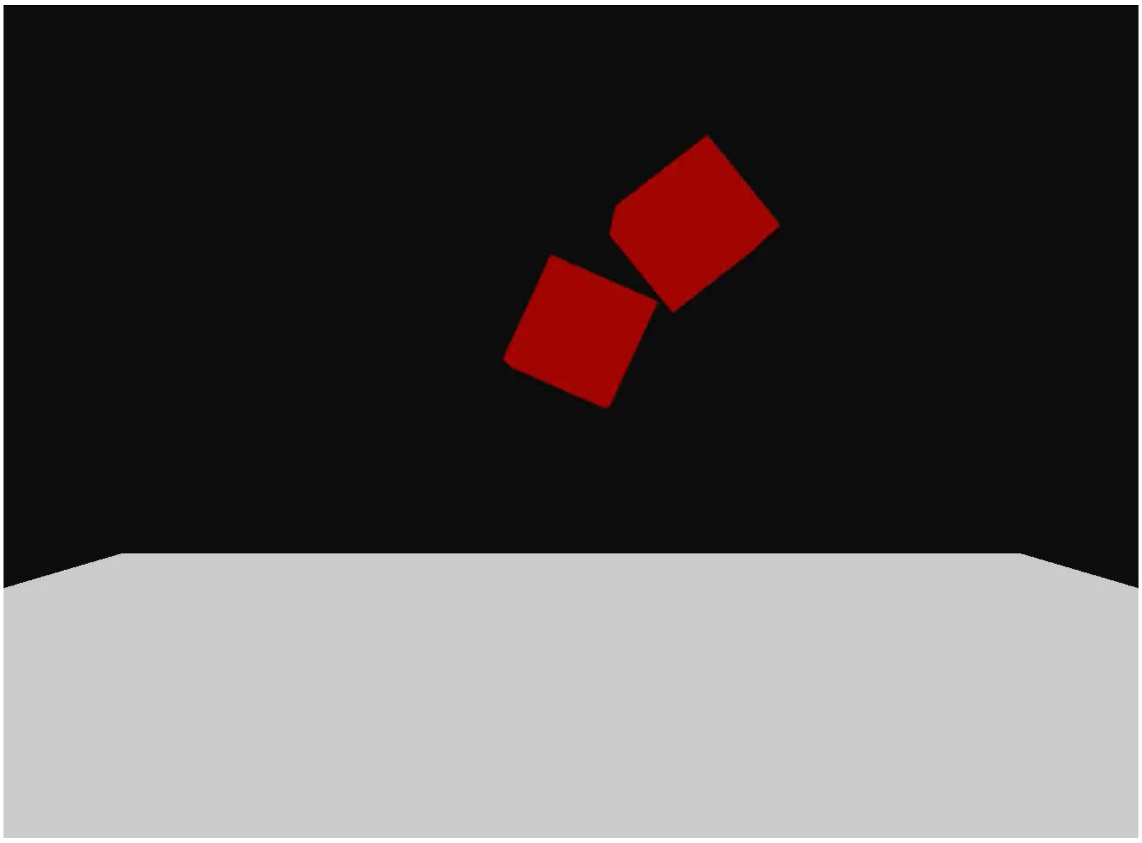

Figure 1. Initial state.

In the beginning of the scene depicted in Figure 1, two blocks are placed in the air and some initial forces and torque are applied to make the results more interesting. Figure 2 shows the continuation of the simulation based on this initial setup.

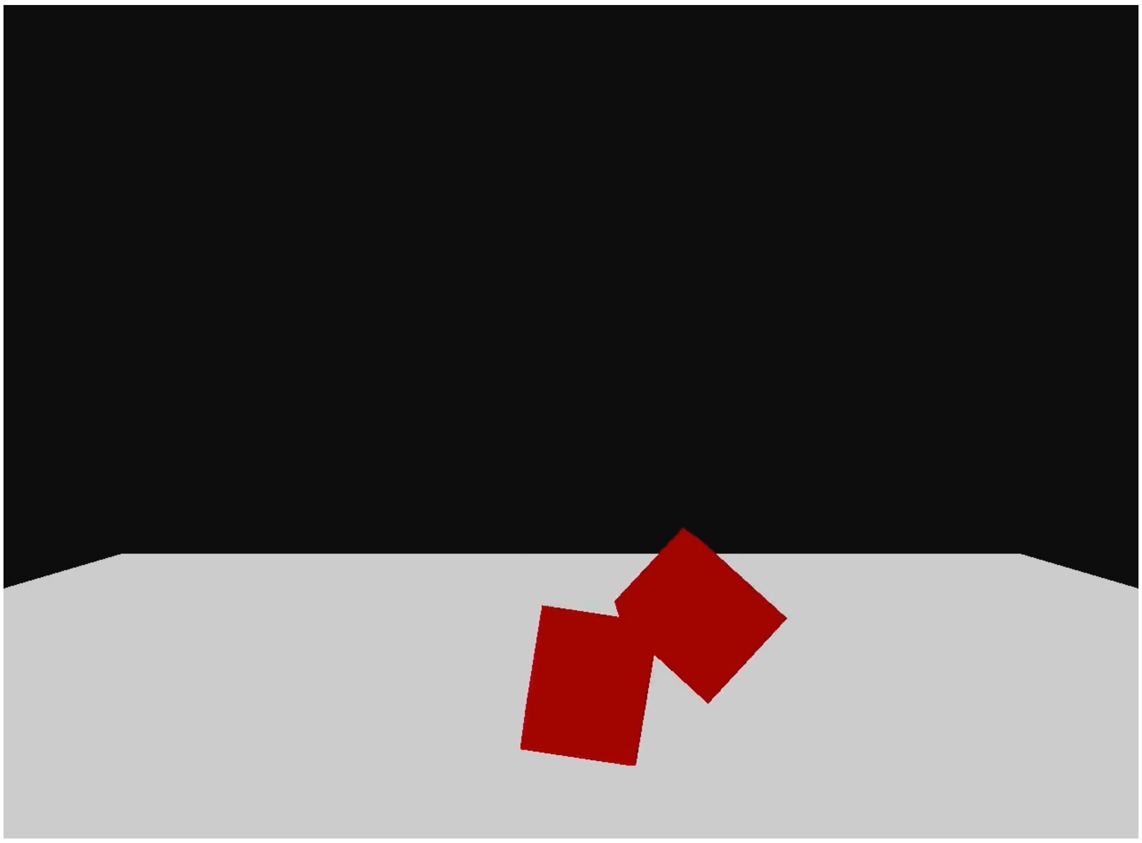

Figure 2. First block bouncing of ground while the second block is colliding with it.

Under the constant gravity, the blocks depicted in Figure 1 fall until they make contact with the ground as shown in Figure 2. Shortly after the first block makes contact with the ground it also makes contact with the second block. The result of the collision resolution is shown in Figure 3.

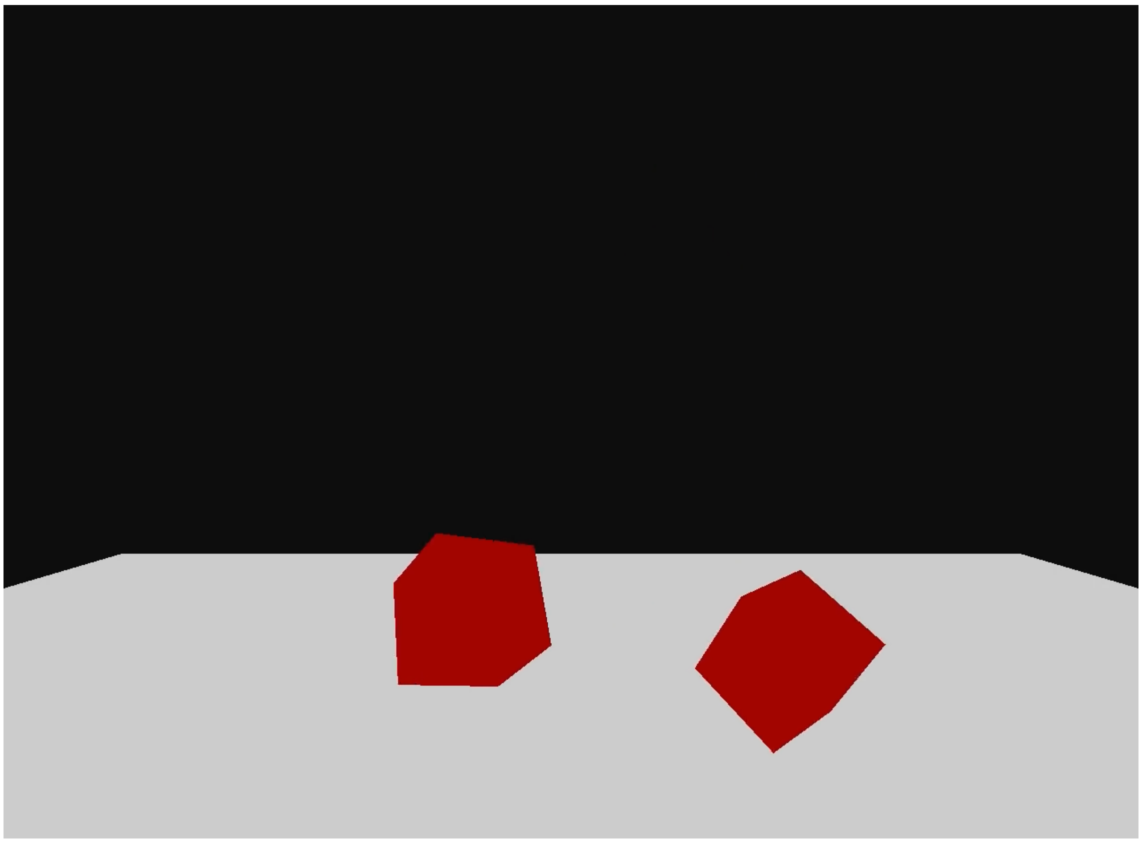

Figure 3. The two blocks bouncing away from each other after the collision.

Figure 3 shows the blocks bouncing away from each other after the collision shown in Figure 2. After the collision in Figure 2 the first block to hit the ground bounces back towards the ground after being hit by the second block. The second block bounces directly away after this collision as does the first block after coming into contact with the ground plane again. Figure 4 shows the end result of these collisions.

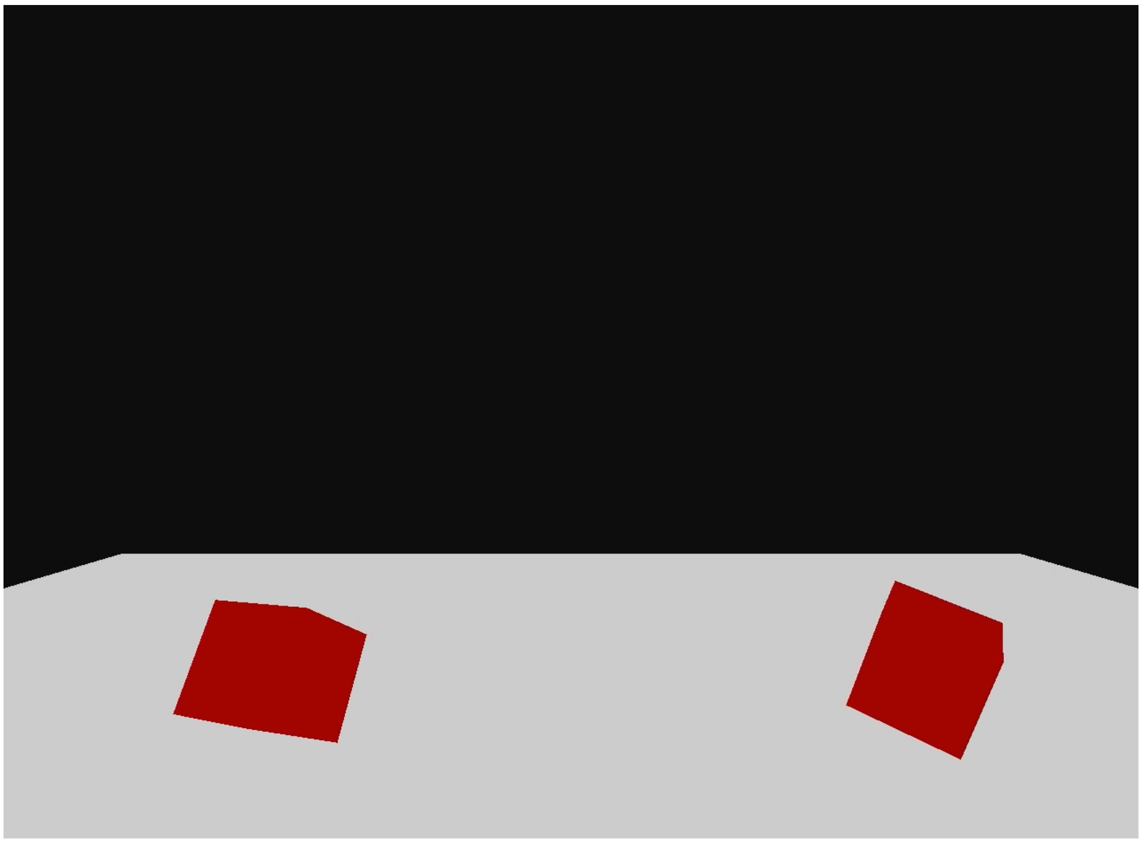

Figure 4. The two blocks colliding with the ground again.

Figure 4 shows both of the blocks coming into contact with the ground again after the bounce shown in Figure 3. After this, both blocks settle on the ground plane.

6 Conclusions

While the developed engine pales in comparison to the existing solutions, the primary goals of the project were still achieved. Some compromises were made at the end of the project and it was not completely, “productized”, since the main purpose was already achieved. The experience of implementing the math and algorithms from scratch offered deep insight into usage and potential issues of physics simulations. Additionally, some intuitive knowledge and better appreciation on how to integrate any of the existing solutions to a project were gained. Also based on this project it would be possible to implement a custom-built physics engine for specific game or demo projects.

One related but largely unexplored area in context of this project is the collision detection. For future continuation of the project in the spirit of learning it would be an interesting task to also implement the collision detection algorithms from scratch. Moreover, the other methods and algorithms for resolving collisions could be investigated in more detail to develop more in depth understanding of collision resolution. Aside from the currently implemented algorithm no other algorithms were studied, which left the in-depth understanding of the algorithm lacking.

References

[1] J. Gregory, Game Engine Architecture, 2nd ed., CRC Press, 2013.

[2] [Online]. Available: https://www.havok.com/. [Accessed 11 3 2017].

[3] [Online]. Available: http://bulletphysics.org. [Accessed 11 3 2017].

[4] [Online]. Available: http://www.geforce.com/hardware/technology/physx. [Accessed 11 3 2017].

[5] "Box2D," [Online]. Available: https://box2d.org/. [Accessed 2 11 2019].

[6] I. Millington, Game physics engine development, 2nd ed., CRC Press, 2010.

[7] G. Fiedler. [Online]. Available: http://gafferongames.com/game-physics/fix-your-timestep/. [Accessed 19 3 2017].

[8] J. Kåhrström. [Online]. Available: http://kahrstrom.com/gamephysics/2011/08/03/euler-vs-verlet/. [Accessed 19 3 2017].

[9] G. Fiedler. [Online]. Available: http://gafferongames.com/game-physics/integration-basics/. [Accessed 19 3 2017].

[10] [Online]. Available: http://wiki.vdrift.net/index.php?title=Numerical_Integration. [Accessed 19 3 2017].

[11] F. Boesch. [Online]. Available: http://codeflow.org/entries/2010/aug/28/integration-by-example-euler-vs-verlet-vs-runge-kutta/. [Accessed 19 3 2017].

[12] "Semi-implicit Euler method," [Online]. Available: https://en.wikipedia.org/wiki/Semi-implicit_Euler_method. [Accessed 6 11 2019].

[13] "Verlet integration," [Online]. Available: https://en.wikipedia.org/wiki/Verlet_integration. [Accessed 7 11 2019].

[14] C. Ericson, Real-time collision detection, CRC Press, 2005.

[15] 2010. [Online]. Available: http://www.dyn4j.org/2010/01/sat/. [Accessed 1 4 2017].

[16] 2010. [Online]. Available: http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/. [Accessed 1 4 2017].

[17] 2010. [Online]. Available: http://www.dyn4j.org/2010/05/epa-expanding-polytope-algorithm/. [Accessed 2 4 2017].

[18] [Online]. Available: http://www.rustycode.com/tutorials/convex.html. [Accessed 1 4 2017].

[19] [Online]. Available: https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure. [Accessed 2 4 2017].

[20] A. Petersen, 2012. [Online]. Available: http://buildnewgames.com/broad-phase-collision-detection/. [Accessed 2 4 2017].

[21] "Coefficient of restitution," [Online]. Available: https://en.wikipedia.org/wiki/Coefficient_of_restitution. [Accessed 14 11 2019].

[22] "Impulse of Force," [Online]. Available: http://hyperphysics.phy-astr.gsu.edu/hbase/impulse.html. [Accessed 14 11 2019].

[23] "Git," [Online]. Available: https://git-scm.com/. [Accessed 2 11 2019].

[24] "GitHub," [Online]. Available: https://github.com/. [Accessed 2 11 2019].

[25] "GitLab," [Online]. Available: https://about.gitlab.com. [Accessed 11 2 2019].

[26] "Bitbucket," [Online]. Available: https://bitbucket.org. [Accessed 2 11 2019].

[27] "CMake," [Online]. Available: https://cmake.org/. [Accessed 2 11 2019].

[28] "Comparing Meson with other build systems," [Online]. Available: https://mesonbuild.com/Comparisons.html. [Accessed 14 11 2019].

[29] "How to structure your project," [Online]. Available: https://cliutils.gitlab.io/modern-cmake/chapters/basics/structure.html. [Accessed 14 11 2019].

[30] "Epoxy," [Online]. Available: https://github.com/anholt/libepoxy. [Accessed 11 2 2019].

[31] "GLFW," [Online]. Available: https://www.glfw.org/. [Accessed 11 2 2019].

[32] "Assimp," [Online]. Available: http://www.assimp.org/. [Accessed 11 2 2019].

[33] "GLM," [Online]. Available: https://glm.g-truc.net. [Accessed 2 11 2019].

[34] "FCL," [Online]. Available: http://gamma.cs.unc.edu/FCL. [Accessed 2 11 2019].