3D Rigid Body Physics Simulation

Olle Grahn, Marcus Lilja, Jonathan Bosson, Torsten Gustafsson

March 13, 2015

Abstract

This report explains the physics used in the simulation of rigid body collisions. The explanation of how the collisions are handled includes linear algebraic and 3D graphics concepts. There is also a description of how the simulation and user interaction is implemented in a 3D graphics environment through C++ with OpenGL.

Contents

Chapter 1

Introduction

1.1 Background

Physics simulations are used in many computer graphic systems today such as video games, simulators, 3D animation software and more. When simulating physics, a *physics engine* is used for simulating real life physics on objects within a scene. For practical purposes, simplifications concerning the physical models are made. Examples of such would be what discrete step method to use, or how accurate the simulated physics needs to be. The principal assumption of a rigid body simulation is that objects do not deform during collision and that changes in velocity during collision are instant.

1.2 Purpose

The purpose of this report is to describe a physics engine developed as part of the course TNM085, *Modelling Project*, at Linköping University. The engine will be able to simulate rigid body movement and collision in 3D. The simulation must also be physically accurate, therefore attributes like mass, inertia and friction will have to be taken into account. The geometric shapes that will be implemented are planes, spheres and boxes. The engine should also be able to render the simulation in real time. The engine will be implemented in C++ with the OpenGL library.

Chapter 2

Physics

2.1 Physics in 3D

The representation of an object in 3D space requires a coordinate vector **x** and a rotational axis **w**. Both the coordinate vector and rotation axis must belong to 3D space.

$$\mathbf{X} = (x, y, z) \in R^3 \tag{2.1}$$

$$\mathbf{w} = (u, v, w) \in R^3 \tag{2.2}$$

The translational velocity and acceleration are also represented by vectors. These vectors have a direction and a magnitude.

$$\mathbf{V} = (v_1, v_2, v_3) \quad magnitude = |V| \tag{2.3}$$

$$\mathbf{A} = (a_1, a_2, a_3) \quad magnitude = |A| \tag{2.4}$$

The orientation, rotational velocity and rotational acceleration are described differently from the translational movement. One method is to represent the rotational velocity as the magnitude of the rotational axis. Another method is to extend the rotational axis with a fourth coordinate describing the rotational states.

The different methods have the rotational state in common which is described with a vector and a scalar. The vector describes the direction of the rotation and the scalar describes the state of the rotation.

2.2 Translational Movement

If **X** is the position of the objects center of mass, then the velocity of the center of mass is the derivative of its position.

$$d\mathbf{X}(t)/dt = \mathbf{V}(t) \tag{2.5}$$

The acceleration of the center of mass is defined in the same way.

$$d\mathbf{V}(t)/dt = \mathbf{A}(t) \tag{2.6}$$

Once these properties have been given, the momentum of an object and the forces acting on the object can be calculated.

The translational momentum of an object is equal to its mass times its velocity.

$$\mathbf{P}(t) = \mathbf{V}(t) * m \tag{2.7}$$

The momentum of an object is important when the object collides with another object. This is further explained in chapter 4.

The net force acting on an object is equal to the derivative of its translational momentum. Because of this the net force can also be described as a product of the objects acceleration and its mass.

$$\mathbf{F}(t) = d\mathbf{V}(t)/dt * m \tag{2.8}$$

Equation 2.8 is useful when several forces are acting on an object and have to be summarized. There could for example be a gravitational force and a wind force acting on the same object simultaneously.

2.3 Rotational Movement

If θ(t) is the angle with which a particle has rotated around its rotational axis, the derivative of this angle describes the rate at which this angle changes.

$$d\theta(t)/dt = \omega(t) \tag{2.9}$$

The derivative of the angular velocity describes the angular acceleration of the particle

$$d\omega(t)/dt = \alpha(t) \tag{2.10}$$

These three properties describes how fast particles are rotating around their rotational axes, how fast they are going to rotate around it and how much they have rotated around it.

If *p* is a point on a rigid body located a certain distance away from the center of mass and a force is applied to this particle, a torque will be created. Take the vector **r** that goes from the

center of mass to the point, and assuming that the applied force is perpendicular to this vector, the resulting torque will be calculated according to equation 2.5.

$$\tau = \mathbf{r} \times \mathbf{F} \tag{2.11}$$

The net toruqe of all applied forces will be parallell with the rotational axis.

$$\tau_{net} = \mathbf{I} * \alpha \tag{2.12}$$

Applying a force to this point will result in an increasing velocity and this velocity will create an angular momentum. This angular momentum is calculated in the same way as torque.

$$\mathbf{L} = \mathbf{r} \times \mathbf{P} \tag{2.13}$$

2.4 Impulse Calculation

An impulse is a force applied over a certain time interval.

$$\mathbf{J} = \int_{t_1}^{t_2} \mathbf{F} dt = \delta \mathbf{P} = m * \mathbf{V_2} - m * \mathbf{V_1} \tag{2.14}$$

Impulses and theory about them is very important when solving collision problems. This will be discussed further in chapter 4.

Chapter 3

Discrete Simulation

It is possible to describe all physical models used within this project as differential equations. Thus, all equations can be solved analytically by using indefinite integration. However, this is not a suitable method for simulation purposes.

In simulations, the physical models are discretized with respect to time and numerical methods are used to approximate the solution of the differential equations. Because the model is discrete and not continuous, a time step *h* must be used to calculate each discrete state of the model. The step time may be fixed or variable depending on which simulation method is used.

There are two general approaches concerning numerical methods, *explicit* and *implicit* methods. The future state of an explicit method is approximated from its present state. In an implicit method a solution is found by solving an equation including both the present and the future state. This simulation uses explicit methods since the future states are not known at the present state.

3.1 The Euler Method

One of the most basic and intuitive methods of numerically approximating a solution for a model is using the *Euler Method* [1], or more specifically the *explicit Euler Method*. In the explicit Euler Method, the future value is approximated by stepping forward in the direction of its derivative. The future value Y_{n+1} is calculated from (3.1) where Y_n represents the present state, h is the step time and $f(t_n,y_n)$ is the derivative.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{3.1}$$

3.2 The Runge-Kutta Method

Since the Euler Method is first order, it is not the most accurate approach. For more complex systems and certain step sizes, instability is certain to ensue. A more accurate and stable numerical method is the *Runge-Kutta Method* [2]. There are different variations of the Runge-Kutta method, but this project mainly focuses on the standard *RK4* method.

In the *RK4 Method*, four derivatives are calculated and added together by a weighting function as seen in (3.2). Y_{n+1} is then calculated by stepping forward with time step *h* in the direction of the weighting function.

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.2}$$

The first k-element k_1 is based on the slope at the beginning of the interval using *y*, just like in the explicit Euler Method.

$$k_1 = f(t_n, y_n) \tag{3.3}$$

The second element, k_2 is based on the slope in the mid-point of the interval using the previous k_1 value.

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{3.4}$$

Likewise, the thind element k_3 is based on the slope in the mid-point but with k_2.

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \tag{3.5}$$

The final element k_4 is the increment based on the slope at the end of the intervall using k_3.

$$k_4 = f(t_n + h, hk_3) \tag{3.6}$$

This method is less prone to instability and gives a more accurate solution. However it takes considerably more steps and longer time to approximate a solution. For this project it is imperative to calculate the solutions as fast as possible since hundreds of objects are expected to interact with each other in real time. The results must of course be stable and fairly accurate, but the user will not notice the differences in accuracy between the Euler method and RK4. Therefore, only the explicit Euler method is used in the simulation.

Chapter 4

Collision

In this project, collisions are handled differently depending on what objects are colliding. Three object types have been defined; spheres, boxes and planes. It is assumed that planes are immobile and are therefore static rigid bodies.

This means that collision detections and collision responses are calculated in several different ways. The calculations performed when two boxes collide will for example be different from the calculations performed when a box and a sphere collide.

4.1 Collision Detection

4.1.1 Collision Detection Between Spheres

It is simple to check whether or not two spheres have collided. Take the center of mass position of one sphere and subtract it from the position of the other sphere. Then check the length of the resulting vector and compare it to the radius of one sphere added to the radius of the other sphere. If the length of the vector is shorter than the two radiuses then a collision has occured.

$$\mathbf{p_1} - \mathbf{p_2} = \mathbf{n} \tag{4.1}$$

$$|\mathbf{n}| < (radius_1 + radius_2) \rightarrow collision \tag{4.2}$$

4.1.2 Collision Detection Between Boxes

It is not difficult to visually detect if two boxes have collided, but to determine it in a mathematical representation of 3D space is more complex. There are several ways to implement this detection and they all have their respective pros and cons. [5] and [3] give a good description of some of these methods.

In this project two different methods were used.

Comparing vertex coordinates in a local basis

All objects in the scene have a unique translational and rotational matrix that determine the location of the object in a certain time instant. Both of these matrices are derived from the mathematical representations of the object's position and orientation.

This means that an object is created with its center of mass in the origin of the unit basis and the vertex coordinates that describe the shape around it. The object is then transformed with the matrices to its new position in the unit basis.

To check if the coordinates of one box is within another, the unit basis coordinates of the first box must be transformed to the local basis of the second box.

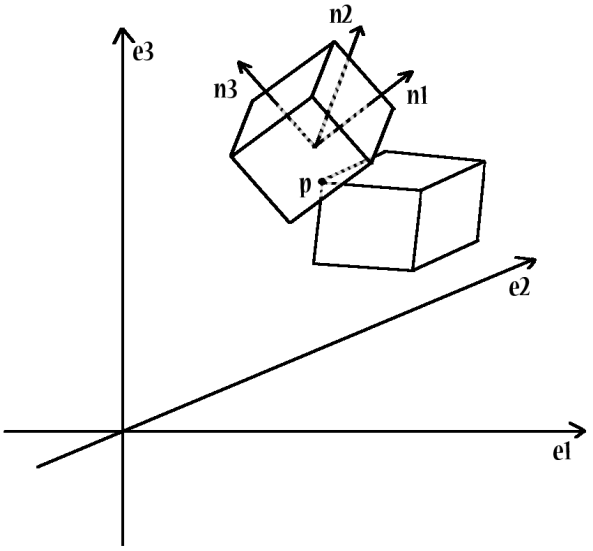

Figure 4.1: Vertex points of one cube is transformed to the local basis of another. If this coordinate is within the cube dimensions a collision has happened.

When the coordinates are transformed to the local basis system of the second box it is simply a matter of comparing these coordinates to the dimensions of the second box. If all three coordinates are within the dimensions of the second box a collision has occured.

$$Unit \quad basis \quad \mathbf{e_1} = (1,0,0) \quad \mathbf{e_2} = (0,1,0) \quad \mathbf{e_3} = (0,0,1) \tag{4.3}$$

$$New \quad basis \quad \mathbf{n_1} = \mathbf{R}_1 * \mathbf{e_1} \quad \mathbf{n_2} = \mathbf{R_1} * \mathbf{e_2} \quad \mathbf{n_3} = \mathbf{R_1} * \mathbf{e_3} \tag{4.4}$$

Vertex coordinate of second box in local basis of first box;

$$\mathbf{p_1} = \begin{Bmatrix} \text{u.x} & \text{u.y} & \text{u.z} & 0 \\ \text{v.x} & \text{v.y} & \text{v.z} & 0 \\ \text{w.x} & \text{w.y} & \text{w.z} & 0 \\ 0 & 0 & 0 & 1 \end{Bmatrix} * \begin{Bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{Bmatrix} * \mathbf{p_2} \tag{4.5}$$

$$if(-1 < \mathbf{p_1}.x < 1 \; and \; -1 < \mathbf{p_1}.y \; < 1 \; and \; -1 < \mathbf{p_1}.z < 1) \rightarrow collision \tag{4.6}$$

Both boxes have to be compared against each other. The vertex coordinates of the first box must be checked in the local basis of the second box and the coordinates of the second box must be checked in the local basis of the first box. If a vertex coordinate is penetrating the box this coordinate is also used as the collision point. This approximation is acceptable as long as the simulation steps are small enough.

The edges of the boxes can also collide. This collision detection is determined by creating vectors between the vertex coordinates of the boxes and checking if these vectors penetrate the planes of the other box. As of the writing of this report this check has not yet been implemented.

Separating Axis Theorem (SAT)

One method used to detect if two objects collide is the Separating Axis Theorem [4]. The idea of the method is projecting each vertex point of both objects on the line that crosses both objects' center. Then, if one or more of their respective vertex positions overlap the method will call it a collision. To get the exact position of the collision, the point between the two vertex positions are used. If more than one point has crossed the other object, a different approach is used to get the collision-point, using an average of the points.

This method is a simplified way to solve a complex problem, and its accuracy depends on the amount of vertex positions on each object. With few points the accuracy is relatively low. Figure 4.2 below shows an example of this. Objects with high amounts of vertex positions are more accurate in detecting collisions, but will also increase the computation time for the detection.

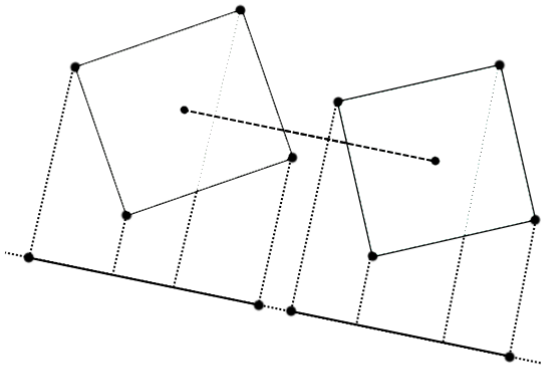

Figure 4.2: Vertex positions are projected to a plane and then compared with each other.

4.1.3 Collision Detection Between Planes and other Objects

This collision detection is done by transforming the object coordinates to the local basis of the plane and then comparing these transformed coordinates to the dimensions of the plane.

This check is performed in almost the exact same way as in section 4.1.2. The difference being that n_1 is now the normal of the plane. If any coordinates are negative along the n_1 axis and within the plane dimensions along the other axes, a collision has occurred.

$$if(\mathbf{p_1}.x < 0 : and : -1 < \mathbf{p_1}.y < 1 : and : -1 < \mathbf{p_1}.z < 1) \rightarrow collision \tag{4.7}$$

4.2 Collision Response

When bodies collide they obviously apply forces to each other. Just applying a force is not enough when two rigid bodies collide however. A force cannot instantly change the velocity of object which leads to further interpenetration. To avoid that the objects move inside of each other their velocities must be changed instantaneously.

This instantaneous change can be approximated as an impulse. This collision impulse will be a very large force acting over a very short amount of time.

The collision impulse can be calculated in many different ways depending on how realistic the simulation has to be.

Since collisions in this project are handled on a per object basis, the impulse calculations differ somewhat depending on the objects that collided.

4.2.1 Collision Response Between Spheres

The sphere to sphere collision response has been simplified to make the simulation run faster. The simplification that has been made is that the rotational momentum at the time of collision does not affect the translational momentum.

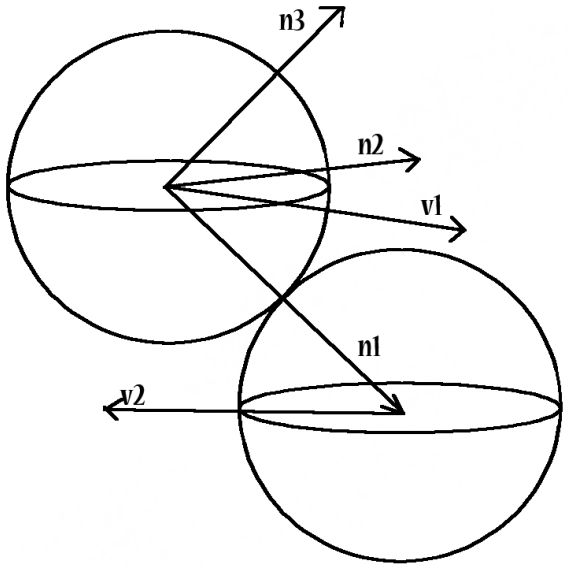

Figure 4.3: Two spheres colliding and the vectors necessary to calculate the collision response

As two spheres collide the translational velocities of the spheres will change along the n_1 vector that can be seen in Figure 4.3 [8]. n_1 goes through both spheres at their center of mass. If two more vectors are created, both perpendicular to n_1 and each other, a new local basis is obtained according to Figure 4.3.

Changing the existing velocity vectors to this basis gives us the translational component alongn_1 and the rotational component along n_2 and n_3.

The new translational velocities along n_1 are calculated according to equations 4.7 and 4.8.

$$\mathbf{v_{1a}}.x = (m_1 * \mathbf{v_{1b}}.x + m_2 * \mathbf{v_{2b}}.x + m_2 * e * (\mathbf{v_{2b}}.x - \mathbf{v_{1b}}.x))/(m_1 + m_2) \tag{4.8}$$

$$\mathbf{v_{2a}}.x = (m_1 * \mathbf{v_{1b}}.x + m_2 * \mathbf{v_{2b}}.x - m_2 * e * (\mathbf{v_{2b}}.x - \mathbf{v_{1b}}.x))/(m_1 + m_2) \tag{4.9}$$

Here $\mathbf{v_{1a}}$ and $\mathbf{v_{2a}}$ are the velocity vectors in the new base after collision and $\mathbf{v_{1b}}$ and $\mathbf{v_{2b}}$ are the vectors before collision. m_1 and m_2 are the masses of the two spheres.

The change in rotational velocity is calculated according to equations 4.9 and 4.10.

$$\mathbf{w_{1a}} = \mathbf{n_1} \times ((\mathbf{n_1} \times \mathbf{w_{1b}}) + (0, \mathbf{v_{2a}}.y, \mathbf{v_{2a}}.z)) \tag{4.10}$$

$$\mathbf{w_{2a}} = \mathbf{n_1} \times ((\mathbf{n_1} \times \mathbf{w_{2b}}) - (0, \mathbf{v_{1a}}.y, \mathbf{v_{1a}}.z)) \tag{4.11}$$

4.2.2 Collision Response Between Boxes

When the collision point and collision normal $\mathbf{s_N}$ have been determined according to Figure 4.4 the collision response can be calculated. $\mathbf{r_1}$ and $\mathbf{r_2}$ are the vectors that go from the center of mass to the collision point.

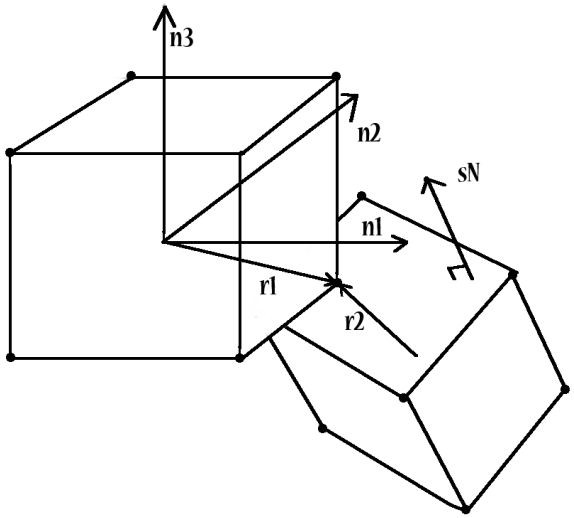

Figure 4.4: Vertex positions are projected to a plane and then compared with each other.

The new translational and rotational velocities are then calculated with equations 4.11 and 4.12. The velocity vector notations are the same in these equations as they were in equations 4.14 and 4.15.

$$\mathbf{v_{1a}} = \mathbf{v_{1b}} + j * \mathbf{s_N}/m_1 \quad \mathbf{w_{1a}} = \mathbf{w_{1b}} + \mathbf{I_1}^{-1} * (\mathbf{r_1} \times j * \mathbf{s_N}) \tag{4.12}$$

$$\mathbf{v_{2a}} = \mathbf{v_{2b}} - j * \mathbf{s}/m_{\mathrm{N2}} \quad \mathbf{w_{2a}} = \mathbf{w_{2b}} - \mathbf{I_2}^{-1} * (\mathbf{r_1} \times j * \mathbf{s_N}) \tag{4.13}$$

I is the inertia tensor and *j* is the created impulse. j is calculated in equation 4.13 [5].

$$j = \frac{-(1+e) * v_{\mathrm{r}}}{1/m_1 + 1/m_2 + \mathbf{s_N} * ((I_1{}^{-1} * (\mathbf{r_1} \times \mathbf{s_N})) \times \mathbf{r_1}) + \mathbf{s_N} * ((I_2{}^{-1} * (\mathbf{r_2} \times \mathbf{s_N})) \times \mathbf{r_2})} \tag{4.14}$$

v_{r} is the relative velocity between the two colliding points of the object.

$$v_{\mathrm{r}} = \mathbf{s_N} * (\mathbf{p_{v1}} - \mathbf{p_{v2}}) \tag{4.15}$$

The velocity of a point on the object is simply the translational velocity of the object added to the rotational velocity of the point.

$$\mathbf{p_v} = \mathbf{v} + \mathbf{w} \times \mathbf{r} \tag{4.16}$$

4.2.3 Collision Response Between Planes and other Objects

This collision response is calculated in the same way as in section 4.2.2. The difference being that no velocities are calculated for the plane and the plane velocities are always set to zero.

Chapter 5

Implementation

5.1 C++ with OpenGL

The system was implemented in C++ with *OpenGL* as its graphical interface. The library GLFW was used together with OpenGL to render windows and handle user input, such as mouse and keyboard input. The library *OpenGL Mathematics*, *GLM*, was used for handling vector and matrix related calculations.

The simulations were calculated using the main CPU. By using OpenGL all graphics related calculations such as light shading and camera perspectives could be calculated using the graphics card, which is immensely fast in comparison to calculations done on the CPU. Thus the graphics would not slow the simulations done on the CPU down.

A basic lighting shader was written in *GLSL* that illuminates the scene from a single light source placed over the scene. The *Phong shading interpolation* method together with the *Phong reflection* model were used to shade all objects in the scene. The Phong reflection model consists of three lighting components: ambient, diffuse and reflection.

For the resulting program, a system for making different scenes was developed. A scene whithin the context of this program is a physics environment that can consist of several objects. These objects are either immovable *static rigid bodies* such as walls and slopes etc. or *dyamic rigid bodies* such as spheres and boxes. A couple of different demo scenes have been made, each with different properties such as varying mass on dymanic rigid bodies and different wall placements.

5.2 Interaction in 3D

In the final program, the user may interact with the scenes in several ways. By pressing the numeric keys the user can select different scenes. At any moment, the user can add or remove dynamic rigid bodies to the scene in real time. There is no limitation on how many objects that can be added to a scene.

A camera system has also been implemented, allowing the user to orbit the viewport around the scene to get a desired perspective. The camera can be rotated freely about the origin and

can be moved closer or futher away from the scene.

The keyboard can be used to apply global forces to all abjects in a scene. The mouse can be used to select and move individual dynamic rigid bodies. The mouse interaction is done through a conversion from the 2D screen to the 3D world coordinates.[6]

The pixel position of the cursor is first captured and then remapped to values between [-1, 1]. By adding a third axis with the same range, can the vector be transformed with the inverse of the projection matrix used to set up the viewport. This will generate the cursor eye coordinates. A cursor ray from the camera towards the selected 3D world coordinates is received by multiplying these coordinates with the inverse of the view matrix. The ray is later used to measure the distance between the cursor and an objects' center. This is done by taking the object vector seen in figure 5.1 and calculating its orthogonal projection with the cursor ray. If the distance is shorter than a fixed threshold, the object is seen as selected and can be interacted with.

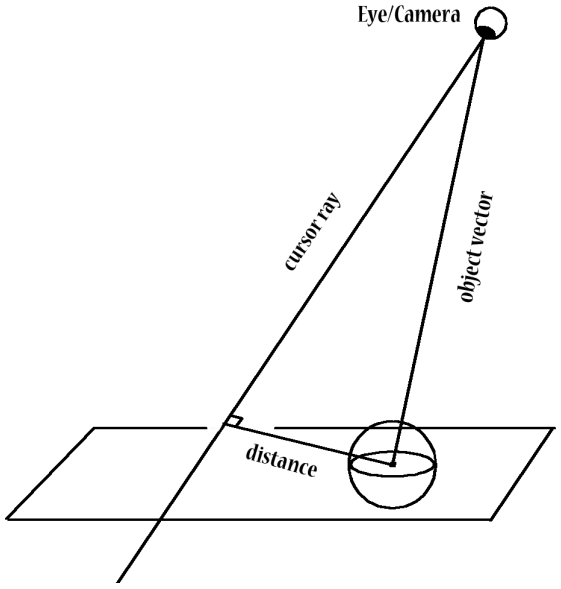

Figure 5.1: Mouse picking with ray casting.

Chapter 6

Results

The project have generated a physics engine able to detect and solve collisions between objects. Both translational and rotational collisions are handled in an effective way. Other physical properties handled are non-elastic collisions and kinetic friction. The simulations are fast enough for a standard computer in 2015 to handle around a hundred objects at once.

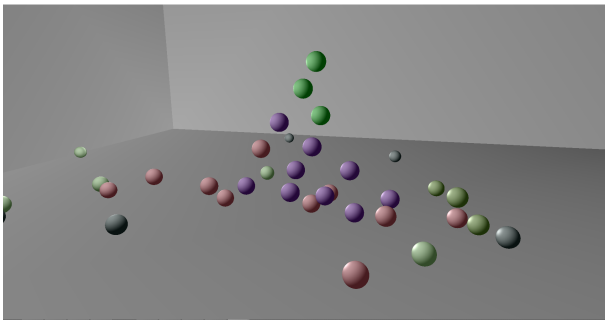

Figure 6.1: A simulation of spheres in a scene.

6.1 Simulation

All properties that are required to represent the translational and rotational movement of a rigid body have been implemented. A representation of gravity has been implemented and users can input global forces in six directions.

The discrete integration of the resulting acceleration is done with the explicit euler method which has been fully implemented.

In regards to collision the following has been implemented;

- Collision detection and responses between spheres have been fully implemented.
- Collision detection between spheres and planes have been fully implemented but only the translational collision response has been implemented.
- The collision detection between boxes has yet to be correctly implemented. The collision response between boxes has been fully implemented.

- Collision detection between boxes and planes have been fully implemented but only the translational collision response has been implemented.

6.2 Graphical Enviroment

A graphical environment for the physics engine has been created in C++ with OpenGL. Different scenes can be displayed and are all lit with one global light source.

The color of the dynamic rigid bodies are currently based on their masses. A heavy body is drawn with a red color and a body with lighter mass is drawn with a blue color.

6.3 Interaction

The user can select which scene should be displayed from a few pre made ones. Dynamic rigid bodies may be added or removed from the scene. The camera can orbit around the scene as well as go closer or further away from the origin.

Global forces can be applied from the keyboard and the mouse can be used for selecting individual objects and manipulating them.

Chapter 7

Discussion

7.1 Simulation

The physics engine uses the simple explicit Euler method for the simulation. Since it has a variable step size that depends on the speed of the CPU, problems may occur when many objects are simulated simultaneously. If the system reaches a low frame rate, the step size will be too big, leading to instability regarding the collision detection. As a result, with a low enough frame rate, the step size might be so big that some objects pass right through each other. A possible solution to this is to change the step size to a fixed value and have the system adapt the rendering of the scene to this step size.

Using more than one CPU thread is also a possible solution. This would free processing power and let the simulation run independently on one thread and the rendering on another. Because of a lack of time and knowledge about threading in C++ this has not yet been implemented.

In the current implementation the light source is not entirely accurate since its position follows the camera instead of staying static.

7.2 Collisions

Since each object types' collision handling are calculated in fundamentally different ways, special cases for each collision type need to be written. Collision types that have been implemented are box to box, box to plane, sphere to sphere and sphere to plane -collisions.

Since this physics engine uses spheres, boxes and planes for simulations, a case for sphere to box collisions needs to be implemented. As of this writing, if a sphere and a box would collide, they would just keep moving through each other which is not intended. This has not been implemented since the other cases took a large portion of the available time.

The collision detection between boxes is not performed in correct way. This is most likely due to numerical errors or an unintended bug. The logic and math behind the collision detection is correct it is simply a matter of correctly translating it to C++ code. Even though it seemed simple it was one of the most time-consuming parts of this project.

Because of the amount of time that was spent on box to box collision proper collision between objects and planes was not implemented within the time-frame of the project. This implementation would not take very long to implement however.

7.3 Interaction

The interaction has to be handled through two dimension input due two the 2D screen. This can make orientation and direction in the 3D world obscure. To avert this, the camera is set up looking towards the center of the plane. The camera can move in angles and zoom in and out, however will it always look towards the same origin.

A crucial aspect to consider of the mouse interaction is towards which plane the manipulation should happen. A cursor ray receives new coordinates in all axis depending on the current camera angle and mouse position. As soon as the camera angle is not aligned with the plane along which manipulations are desired, will the transformation be difficult to control. Therefore by putting constraints on in which directions manipulations can be made can such agitation be circumvented.

Bibliography

[1] L. Ljung och T. Glad. *"Modellbygge och Simulering", p. 378, Studentlitteratur*. 2004

[2] L. Ljung och T. Glad. *"Modellbygge och Simulering", p. 380, Studentlitteratur*. 2004

[3] David M. Bourg, Bryan Bywalec. *Physics for Game Developers, Second Edition*. 2013.

[4] David Eberly. *Intersection of Convex Objects: The Method of Separating Axes*. 2008.

`http://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf`

[5] David Baraff. *Robotics Institute, Carnegie Mellon University*. 1997.

`https://www.cs.cmu.edu/~baraff/sigcourse/notesd2.pdf`

[6] Anton Gerdelan. *Mouse Picking with Ray Casting*. 2015.

`http://antongerdelan.net/opengl/raycasting.html`

[7] Chris Hecker. *Physics, The Next Frontier*, *Physics, Part 2: Angular Effects*, *Physics, Part 3: Collision Response*, *Physics, Part 4: The Third Dimension*. 1996.

`http://chrishecker.com/Rigid_Body_Dynamics`

[8] R Grahn, P - Åjansson. *Dynamik*. 1996.

`http://webstaff.itn.liu.se/~ulfsa40/modproj/mek_grahn_jansson.pdf`