Advanced Game And Simulator Programming Report

Daniel Aaron Salwerowicz

March 4, 2018

Abstract

Purpose of this report is to present my work on developing a C++ based physics simulator, that utilizes GMlib2 and GMlib2Qt.

1 Introduction

In this project I have utilized an in-house geometric modeling library developed by Simulations R&D group at UiT called GMlib2. In addition to that I have used Qt, and C++. Resulting simulator is able to quite accurately depict collisions between spheres and planes, spheres and spheres, one at a time or several concurrent collisions. However there are some flaws with my system that I will highlight in the Analysis and Discussion part of this report.

This report will mostly present how I have implemented system for object collision and visualization using Qt3D engine's aspects. Theory is based on a paper written by my lecturers [1] as well as guides provided by Qt3D development team [2] [3].

2 Methods

I've utilized Qt3D aspect to handle most of the work in simulating my physics engine. I will therefore go into detail describing how my project was constructed. `Scene` creates `GUIApplication` object that contains several `QNode` elements such as `TPSphere` and `TPPlane`. These object are then controlled by `ObjectController` communicating back and forth with back-end equivalent `ObjectControllerBackend`. Class `SimulatorAspect` schedules jobs, that then call methods in `Simulator`, results are propagated to the rest of the system using `ObjectControllerBackend`. Figure 1 visualizes this data flow.

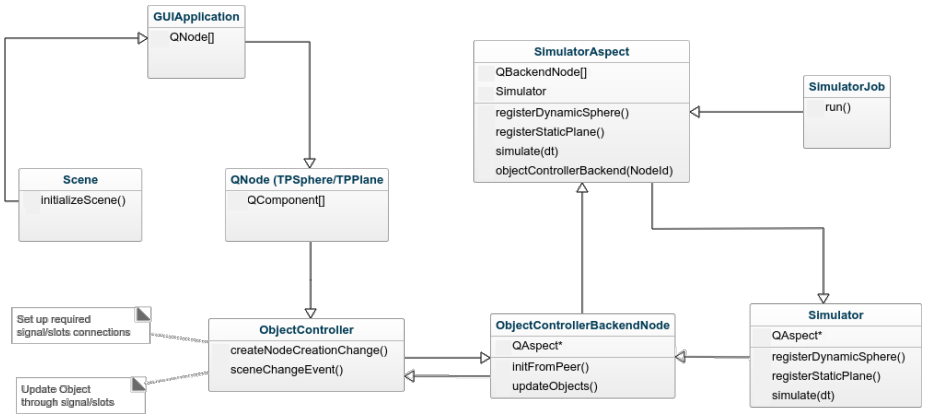

Figure 1: Data flow diagram representing communication between objects in my project

2.1 Predefined classes

`TPSphere` and `TPPlane` are used to represent needed data for spheres and planes in Qt3D used in GUI part of my application. They function as parents for their `ObjectController's` and their data is then converted into structs that are used on the backend. `GUIApplication` sets up those objects and registers them in `Scene`. These classes, besides `ObjectController`, were given to us in GMlib2, GMlib2QT and STET6245TemplateApp projects. They also form so called front-end of my program.

2.2 My classes

2.2.1 ObjectController

It is used to hold basic data about the object it's meant to control. Those are: it's physical properties stored in `PhysObject` struct as well as information about its velocity, type and kind. `createNodeCreationChange` method sets up this struct, where front-end object is converted to struct which will be used in communication with back-end.

2.2.2 ObjectControllerBackend

This class takes pointer to `SimulatorAspect` which it uses to register simulation objects. Registration happens in `initFromPeer` method that takes reference to `QNodeCreatedChangeBasePtr` created in `ObjectController` and uses data in it to register object used in back-end for simulation. It also has a small class `ObjectControllerBackendMapper` that is used to map front-end and back-end elements together.

2.2.3 SimulatorAspect

Besides registering simulation objects and `ObjectControllerBackends` this class does several other things. They are: scheduling of jobs for execution and calling `simulate` method in `Simulator`. `SimulatorJob` a small helper class is used to store timestep value and call `checkCollisions` method in all registered `ObjectControllers`.

2.2.4 Simulator

This is the most important class in this project. `simulate` method thats it contains gives life to the simulation. It starts with initialization which include setting several variables to standard values for each timestep.

`collisionCheck` method detects collisions for each dynamic object with other dynamic objects as well as static object. Those checks are done in helper class `collision_algorithms`, checks are based on calculations presented in detail in [1]. Collisions are sorted by time and made unique, so that if for example two balls collide, only one collision found by algorithms is simulated.

Colliding objects are simulated up to collision in appropriately called method, impact response is calculated and new collision checks are done. This is done for each collsion in a given timestep.

Lastly all remaining dynamic objects are simulated to the end of this timestep and new timestep can be calculated.

2.2.5 Unit tests

Per requirement all collision detection algorithms in my project are covered by unit tests. They ensure that all of the collision results are correct. If I had more time I would implement unit tests for some critical methods in simulation.

3 Results

Result of this project is a simulator that has basic setup for collision detection implemented. I am able to set up several objects in my environment and simulate their movement with quite high accuracy. The biggest problem that I have encountered so far is un-handled singularities. Because of that spheres can phase through planes when their velocity is too low or they are moving almost parallel to the plane in time of collision. Despite that this simulator handles simulation of several concurrent collisions quite well.

State changes were not implemented as I lacked time to do so. First week was spent on setting up the project in Windows environment, which quickly prompted me to switch over to Linux. Understanding Qt3D aspects was another roadblock that significantly slowed down my progress.

As it is, this project is a great, but also a bit flawed foundation for further development. Figure 2 shows simultaneous collision between three balls.

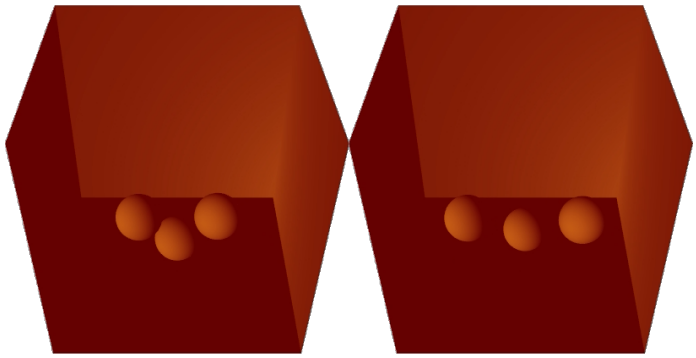

Figure 2: Before and after collision between three balls

4 Analysis & Discussion

This project has been a great learning experience and although resulting program leaves much to be desired I am quite happy with progress I have made thanks to my colleagues.

Analyzing resulting product I can clearly see that I need to handle state changes and some fringe edge cases, which my program does not do at the moment. After implementing those features I would need to work on rolling spheres as well as collision with cylinders and finite planes. Lastly I would like to implement my planned simulation, description of it that can be found in appendix.

5 Acknowledgments

I want to thank Christopher Hagerup for helping and guiding me while I worked on my simulator. He has helped me a lot during development so there's a lot of similarity in our projects and way of thinking. Kent Arne Larsen and Victor Lindbäck have also helped me in my work.

6 Bibliography

[1] Jostein Bratlie, Rune Dalmo, Arne Lakså, and Tanita Fosli Brustad. STE6245 - Course Material, December 2017. [Accessed 2018-03-04].

[2] Sean Harmer. Writing a Custom Qt 3D Aspect - Part 1. Online, available at https://www.kdab.com/writing-custom-qt-3d-aspect/, November 2017. [Accessed 2018-03-04].

[3] Sean Harmer. Writing a Custom Qt 3D Aspect - Part 2. Online, available at https://www.kdab.com/writing-custom-qt-3d-aspect-part-2/, December 2017. [Accessed 2018-03-04].

7 Appendix

7.1 Setting up project

7.1.1 Required packages

Project was set up on an Antergos machine. Since I was using Clang as a compiler I was forced to set up some ignore flags in CMakeLists.txt files for GMlib2, GMlib2qt and Template app. Therefore I recommend using my forked repositories. Following packages need to be compiled and installed with their dependencies so that this project is able to run.

- Latest QtLibrary, at least version 5.6
- Blaze
- Clang, at least version 5.0
- CMake, at least version 3.8
- Ninja
- Gtest
- Google benchmark

7.1.2 Building projects

GMlib2 needs to be built first, building can be done in Qt Creator or through terminal. CMake needs to be set up with following flag `blaze_DIR` which points to cmake directory in blaze installation directory. https://source.uit.no/mormonjesus69420/gmlib2-prototype

GMlib2Qt needs to be built after GMlib2 and point to its cmake directory in `gmlib2_DIR` CMake variable. GMlib2 needs to be set as its dependency. https://source.uit.no/mormonjesus69420/gmlib2qt-prototype

Lastly STE6245TemplateApp is built. CMake will need `gmlib2qt_DIR` variable to point to cmake directory in GMlib2Qt, as well as `benchmark_DIR` to point to cmake directory in benchmark installation directory. GMlib2 and GMlib2Qt are its dependencies. https://source.uit.no/mormonjesus69420/pachinko-machine

7.2 Task Description

My plan is to implement a working simulation of Japanese gambling game called Pachinko. This game resembles the western pinball machine. It sends several steel balls from the bottom of the playing field, which is usually circular in shape. Balls then travel along the track and upon reaching the top of the playing field they start to fall down, bouncing against each other and against the metal pins that stick from the board. Goal itself is to have as many balls

as possible fall down into the designated small cups strewn across the board, several versions include flippers moving randomly making it harder to hit those cups. An example of such board is shown in figure 3.

Figure 3: An example of old fashioned Pachinko machine.

My plan is to simulate that game, though what I'm aiming for is a simplified version which will use a rectangular box with several rows of pins and some receiving boxes on the bottom. A mockup of the working game/simulation is shown in figure 4.

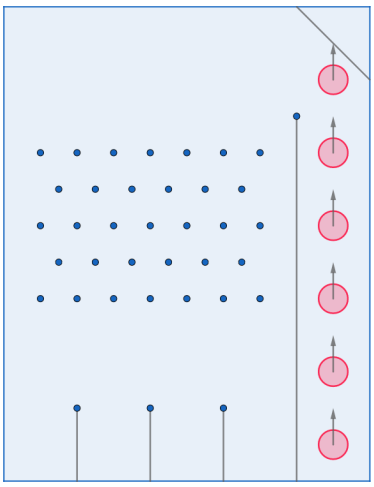

Figure 4: Mockup of the working game with some balls drawn in to show how the game itself starts.