BACHELOROPPGAVE:

Noxplus

FORFATTERE:
Tien Quoc Tran
Håkon Bjørklund
Even Arneberg Rognlien

DATO:
15.05.2015

Sammendrag av Bacheloroppgaven

Tittel:	**Noxplus**
Dato:	15.05.2015
Deltakere:	Tien Quoc Tran Håkon Bjørklund Even Arneberg Rognlien
Veiledere:	Mariusz Nowostawski
Oppdragsgiver:	Suttung Digital
Kontaktperson:	Håkon Bjørklund, hkonbjork@gmail.com, 48 06 23 56
Nøkkelord:	Norway, Norsk
Antall sider:	166
Antall vedlegg:	
Tilgjengelighet:	Åpen

Sammendrag:

Nox er en spillmotor som kan brukes for å lage 2D-baserte spill. Noxplus er en utvidelse av denne spillmotoren og består av to hovedmoduler. Den første hovedmodulen går ut på å implementere muligheten for å spole frem og tilbake i tid mens spillet kjører. For få til dette på en best mulig måte må ulike lagringsmetoder undersøkes og ytelsestestes. Vi ser også på ulike paradokser som kan oppstå ved tidsreise og hvordan det er mulig å oppdage og løse disse i spillsammenheng. Et design er bestemt og implementert for denne modulen. Den andre hovedmodulen går ut på å gi spillmotoren støtte for 3D grafikk, samt simulering av 3D-fysikk. Dette krever at teamet setter seg inn i mange nye temaer rundt både grafikk- og fysikk motorer. Modulen er splittet opp i tre submoduler; lasting av modeller med mulighet for skjelett-animasjoner, simulering av 3D-fysikk og deferred rendering av lys

Summary of Graduate Project

Title:	**Noxplus**
Date:	15.05.2015
Participants:	Tien Quoc Tran Håkon Bjørklund Even Arneberg Rognlien
Supervisor:	Mariusz Nowostawski
Employer:	Suttung Digital
Contact Person:	Håkon Bjørklund, hkonbjork@gmail.com, 48 06 23 56
Keywords:	Game engine, Programming, Time manipulation, 3D, C++
Pages:	166
Attachments:	
Availability:	Open

Abstract:

Nox is a game engine used for creating 2D games. Noxplus is an extension to this game engine and covers two main modules. The first main module is to implement a time manipulation feature and develop new game mechanics, to seek out new exciting ways of gameplay. This requires research on how storing, rewinding and replaying the game world can be done most efficiently. We also look into different time travel paradoxes, and how they can be detected and resolved in a game. This research culminates into the design and implementation of live rewind and replay support in the Noxplus extension. The second module is to give the engine support for 3D. This includes research on game engine components and how implementation of them can be done. This concludes with the design and implementation of three sub modules; model loading functionality with animation, 3D physics simulation of the world and deferred rendering as the chosen rendering method.

Preface

We would like to thank Mariusz Nowostawski for being our supervisor and valuable asset to our Bachelor, helping us with research, feedback and keeping us on track. Thanks to Simon J. R. McCallum and Frode Haug for pointers and advice. Thanks to Suttung Digital for being our employer. Especially Asbjørn Sporaland and Magnus Bjerke Vik for being our contact persons and giving us feedback and help with the Nox engine and the assignment. We would especially like to extend our thanks to Vik for helping with testing for Mac and Linux in the last stretch of the assignment. Thanks to John Taylor for feedback on the thesis.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Description

1.1.1 Background

We started thinking about what we should do for our Bachelor assignment in late 2014. Our initial thought was to create a custom game engine from scratch and create a game from this. After some discussions with Associate Professor Simon J. R. McCallum and Associate Professor Mariusz Nowostawski, we realised that creating an engine from scratch would be a too large assignment and there would be no way to actually get a good grade. Why not use one of the engines already created? What new things would we add to the field?

Then we heard about a bachelor group from 2014 consisting of Magnus Bjerke Vik and Asbjørn Sporaland. During the years before their bachelor at Gjøvik University College they created a game engine, that they now wanted to create a game for as their bachelor assignment. Their engine is today named Nox. The Nox engine is designed to be flexible and expandable to make it applicable for different types of games. It is written in C++ and is highly modular so that changing the subsystem implementations can easily be done without breaking the system. This engine has advantages and disadvantages in comparison to industry game engines such as Unity3D, Torque2D/3D, and Unreal Engine.

Our first encounter with with Sporaland was through the Graphic Programming course on Gjøvik University College. After some discussions with his company, Suttung Digital, McCallum and Nowostawski, we decided on using their engine and add something new to it. The reason is that the engine is still in an early stage and not as complex as many major game engines out there. They have made it open source, and it's also convenient that the creators are so close by. There were quite a few ideas flying around for the Bachelor assignment and in the end we ended up with two. Read more about the choice of assingment and engine in Section 7.6.1.

We started out as a group of 5 people from 3 different areas of study. Two was from Computer Engineering; Tien Q. Tran and Even A. Rognlien, two was from Information Security; Daniel M. Antonsen and Joakim Harbitz and one was from Game Programming; Håkon Bjørklund. Antonsen and Harbitz will have their own thesis focusing on security within games. The other three would further develop the Nox engine. Both groups was using the Nox engine and game engine development in general as a starting point, and the groups was going to be working together on the security related parts of the project. Certain aspects of the two theses was therefore going to be the same. Later we discovered that we would not be working with security and nothing at all with the network of Nox. However another Bachelor group would be working with the network part on the Nox engine. Therefore we found it appropriate to split into two bachelor groups.

1.1.2 Project goals

Result goals

The main intention was to extend the Nox-engine and give it new functionality. Our contribution will be added to the engine and become public to the world, making the engine

more interesting to other developers. The new functionality is the possibility to create 3D-games with time manipulation for use during gameplay. This entails that a developer can create a world in 3D. The time manipulation will work for both physics, animations and components and makes it possible to reverse and forward time during gameplay. We also created a demo that shows the previously mentioned modules potential.

The bachelor thesis can be used for both developing new innovative games, and further development and research of time management in games and simulation.

The demo and engine will together become something that we can add to our portfolio and show our skill in programming and organisation. It will also become something that Gjøvik University College can use in lessons and research purposes and the bachelor thesis can be added to Gjøvik University College's library B.

Effect goals

Suttung:

- Increased functionality for the Nox-engine.
- More attractive/interesting game engine for other game developers.

Gjøvik University College:

- Research and lesson material that other students can explore and use when studying 3D engines and time manipulation in gameplay.

Noxplus:

- Increased knowledge and skills in game engine programming.
- Increased skill in familiarising oneself with other developers' source code.
- Increased skill in handling larger projects.

1.1.3 Audience

Demo Audience

The demo will be used to demonstrate what we have done for Gjøvik University CollegeİIt is also an introduction for other developers to the updated Nox engine's potential. Gjøvik University College can use it as advertisement to show off what their students are capable of after a completed bachelor degree. For us it will be a project we can add to our portfolio when searching for a job.

Engine Audience

The game and engine developers are the targeted audience for the engine and when Suttung decides to merge the project with the Nox engine, it will become an even more powerful development platform with a little explored area of timeline management in 3D.

Thesis Audience

The thesis is created for Gjøvik University College for research and teaching. It will also be of use for students and other developers who are considering using some of the same tools for creating their own engine or developing a game on the Nox engine. See Section 3.3.2 for full overview of the tools used in this project. It will also be of interest for Suttung to use parts from it as a documentation base for the new improvements.

1.2 Scope

1.2.1 Assignment Description

We focused on the 3D aspect of game engines and how to create a module for this for the Nox-engine. We also included a module for timeline manipulation in games. To begin with, we started working on getting familiar with the existing Nox engine, and also looked into other open source engines. This gave us a better overview of the strong and weak sides of Nox and assessed their features in relation to manipulating timelines, see Section 7.6.1 for why we chose Nox. The actual development task involved system design, high level C++ programming and some low level programming, e.g. GPU shaders.

3D rendering

The 3D rendering module was created as an extension to the Nox engine, so that the engine can render both 2D and 3D. This module was required for the Time manipulation module to be started on, as this is based on 3D timeline manipulation and not for 2D. The task was to implement a renderer that renders 3D models and lights. This also required that we researched and implemented a model loader for loading the models. Read more in Section 2.1 to see what sub modules it will contain, and Section 4.1 under the use case *Render World*.

Time manipulation

We classified different scenarios in time manipulation and from that we decided on what we wanted to implement, and how we should implement it in our module. Read about the choice in Section 4.3.5. The time manipulation had to apply for any actor in a 3D world. The implementation of timeline manipulation ran parallel with the 3D integration to make them as suitable as possible with each other.

3D physics

The original version of the Nox engine uses the Box2D physics library that only supports 2D. Our assignment included to research different physics engines and implement one that supports 3D and our other requirements, read about the research and choice in Section 7.3.

1.2.2 Delimitation

We set the following delimitation's for our modules:

Timeline manipulation involved:

- Ability to pause and play.
- Ability to rewind time in a 3D world both backwards and then forward again.
- Detecting and resolving timeline manipulation paradoxes.

3D capabilities involved:

- Loading in 3D models from ".obj", ".dae" and ".md5" files. With support for textures, bone structure and animation for the file types that support it.
- A scene graph to keep track of every actor in the scene and rendering them.
- Basic 3D lighting; directional, point and spot light.
- A submodule for 3D physics.

The demo includes:

A simple scene where the player can interact with many objects that also interacts with each other, and the whole scene can be played backward and forward through the elapsed play time.

1.3 Project organisation

1.3.1 Responsibilities and roles

The group leader was Håkon Bjørklund, his role was to keep an overview of the project, arrange meetings, distribute work and make sure that goals were met. Keeping an overview of the project involved: keeping track of the time schedule, enforce risk avoidance and make sure the group rules were followed.

The person responsible for the infrastructure was Even A. Rognlien. He made sure we followed the planned architectural design and if there were changes to be done to the design, he updated the documents.

Tien Q. Tran was our version control manager and the person to make sure our repository was always structured and set up correctly. If anything went wrong, he became responsible for fixing it. Tran was also responsible for managing and overlooking the integration with Nox and other libraries.

Asbjørn Sporaland and Magnus Bjerke Vik was our contact persons within Suttung Digital. Suttung Digital was our employer and gave us guidance and requirements for the assignment. They also answered any of our questions regarding the Nox engine.

Mariusz Nowostawski was our supervisor and gave us feedback and guidance on both the theses and the project.

1.3.2 Group background and skills

We had a bit of a varied background, Rognlien and Tran were from Computer Engineering while Bjørklund is from Game Programming. Bjørklund also had a Bachelor in Event Production and Interactive Media. We had worked together before on creating two small games for Android; a 3D game using the Google cardboard API with OpenGL ES, and a 2D game where we did not use any advanced graphics library. Bjørklund was the only one who had created a couple of small games for computers in C++ before. Rognlien and Tran had more experience with creating applications. They also had experience with Linux, while Tran in addition had additional experience with OS X. Only Bjørklund had worked on an assignment of this size before, except from a system development task that all three has done. However, we only did the planning, not the development itself.

We had to learn how the Nox engine is built and how it works. In addition we needed to learn how the physics engine and model loader used works, and how to implement them into Nox. All members had experience with C++ except C++11, but we had all been through a course in graphics programming, so we knew OpenGL.

1.3.3 Practises and rules

Our group had to follow the group rules and the project agreement B C. All group members had to try to keep normal work hours, 8 - 16, but this was not enforced as long as everyone showed up for all planned meetings. When someone came up with an idea for new functionality, he had to take it up with the rest of the group, we then decided if it was a good idea and how we would incorporate it into the current plan.

1.3.4 Risk analysis

	Negligible	Minor	Moderate	Significant	Severe
Rare	Low	Low	Low	Medium	Medium
Unlikely	Low	Low	Medium	Medium	Medium
Moderate	Low	Medium	Medium	Medium	High
Likely	Medium	Medium	Medium	High	High
Very likely	Medium	Medium	High	High	High

Table 1: Depicts how we rate risks, out from likelihood and impact.

1.4 Plan for implementation

1.4.1 Software Development Methodology

Rational Unified Process (RUP) is an iterative development methodology that is used for for example software development [1]. We have been using RUP as our main development methodology for the documentation. Because it gave us many artifacts that were relevant for our thesis, read more about them in Section 3.1.1. Scrum and Extreme programming (XP) is both an iterative agile software development methodology for managing product development. We used Scrum combined with Extreme programing for the development process. You can read more about them in Section 3.1.2 and Section 3.1.3 respectively. This provided us with an agile and flexible implementation process.

The main reason we chose RUP and Scrum is because we needed to see what the end product looked like at a regular basis. Since both are iterative development models this gave us an advantage. If we had chosen a development model like the Waterfall model this would not be possible before late into the project [2]. This would have been a problem for us since we needed to see what effects our new code had on the overall project, to be sure that our plans were good and did not destroy the current functionality of the engine. With an iterative process we were also able to uncover risks early in the development phase instead of late. This mitigated risks and was one of the reasons we chose not to use the spiral development model [2]. The other reason is that RUP has a risk analysis artifact we wanted to use. Scrum on the other hand is missing some documentation for developers, but with the RUP model we believed this would make up for it. The XP model does not have adequate documentation and was therefore a bad model for us since we needed documentation for our thesis. However, there is a couple of principles in XP that we wanted to use for our project.

As we were going to spend most of our time in the development phase we chose to use the Scrum roles. Since Bjørklund is the group leader and many of his assignments align with the assignments of the Scrum Master role it was best that he took this role. All three members took the role as the Product owner, handling the backlog and sprints. More about this in Section 3.1.

1.4.2 Project timeline

As mentioned in Section 1.4.1 RUP and Scrum was our main development model. Our Gantt diagram follow the structure of the RUP phases: inception and elaboration, with a construction phase consisting of Scrum sprints. At the end of our Gantt diagram there is a timeline for our thesis. The inception phase's we established the main foundation for our project by setting up the group rules, contracts, project plan and system design. The

elaboration phase started after the system requirements from Suttung was well enough defined to start working on the system design plans. During the 11 sprints of the construction phase, we implemented the system.

1.4.3 Work breakdown structure

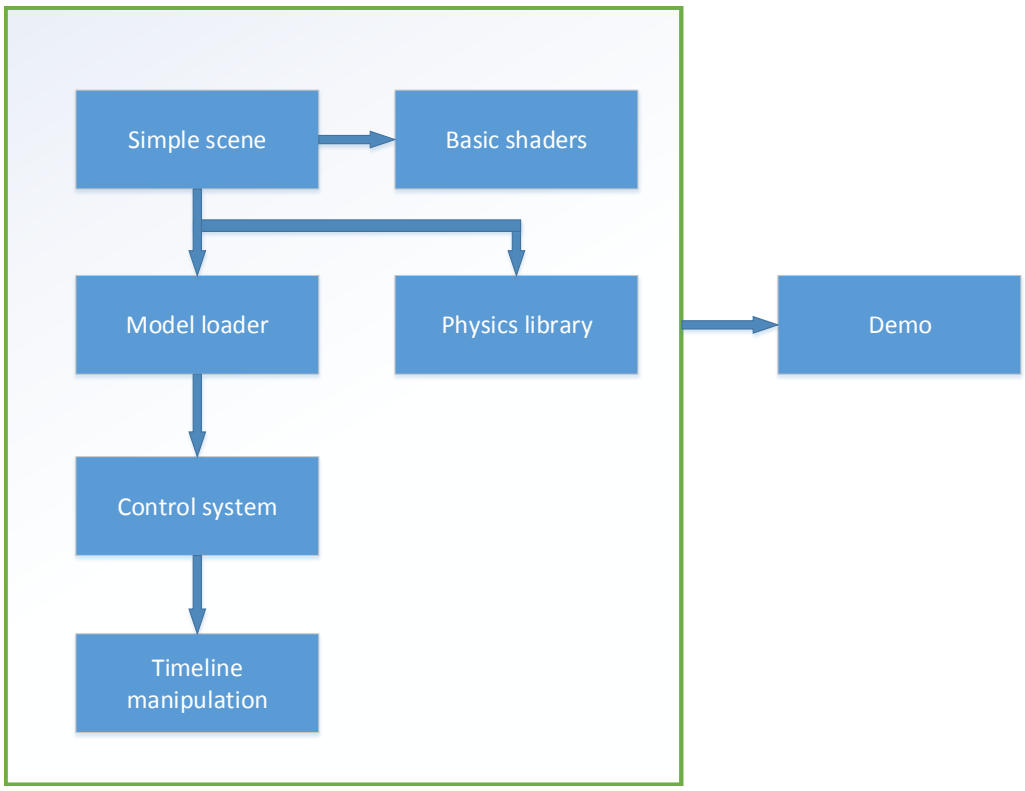


Figure 1: Breakdown of the work.

We wanted to start with a simple, empty scene with a pair of basic shaders. After integrating a model loader, we would load and use some basic 3D models to test the integration of a physics library. The plan was then to add some basic user controls to move the objects around and test their physics. Then we would focus on some simple time manipulation. When all of this were working as expected, the final step would be to create the final demo.

1.5 Terminology

- API - Application Programming Interface.
- GUI - Graphical User Interface.
- IDE - Integrated Development Environment (ex. Visual Studio or Eclipse).
- Json - Javascript Object Notation. Lightweight data interchange format (json.org).
- GUC - Gjøvik University College.
- RUP - Rational Unified Process, development methodology.
- Scrum - An incremental, agile software development methodology. (wiki).
- Artifacts - A document or model created by people involved in a development pro-

cess.
- 3D - Three Dimensions.
- 2D - Two Dimensions.
- VAO - Vertex Array Object
- VBO - Vertex Buffer Object
- Assimp - Open import asset library.
- Bullet - Physics engine.

1.6 Document Structure

The document is structured into eight chapters with Appendixes at the end.

1. Introduction contains the introduction to the thesis, description and the planning of the project.
2. Requirements Specification contains the requirements for the extension.
3. Development Process describes the work flow through the project, how the development methodologies were used and the tools.
4. Design contains the design of the system, the use cases and the flow of the system.
5. Implementation contains how the design and the different modules was implemented.
6. Testing contains how the tests were done and the results.
7. Discussion contains choices that were made under the different modules, including thoughts and ideas.
8. Conclusion contains a summary of what was done and possible future work.

2 Requirements Specification

The assignment is quite large (see Section 1.2) and there were requirements that needed to be concretized before the elaboration phase could start. There were requirements given by Suttung, Gjøvik University College, and some requirements set by ourselves.

2.1 Functionality

For the main modules, defined in 1.4.3, the following functionality requirements were set.

The Scene module includes:

- Rendering in 3D.
- Scene graph for 3D.
- 3D shaders.
- 3D camera.
- API for game developer.

The Model loading module includes:

- Asset loader.
- Static models support.
- Animated models support.
- Texture mapping support.
- API for game developer.

The Physics module includes:

- Physics engine.
- Basic collision shapes (box, sphere, cylinder..).
- Advanced collision shapes (Concave hull, compound..).
- Collision detection with custom made actions.
- API for game developer.

The Timeline manipulation module includes:

- Logging of the game world.
- Logging of custom created components.
- Rewind capability.
- Replay capability.
- Developer created functions for solving conflicts/paradoxes.
- API for game developer.

The Control system module includes:

- Actor controls.
- Camera controls.
- Time controls.
- API for game developer.

The Demo includes demonstrations of:

- Time manipulation.
- Physics.
- Model loading.
- Controls.

2.2 Usability

To be able to use the Nox engine and the extension to its full potential there are requirements to the game developer. The developer must know the programming language C++, since the whole engine and the extension is built on this language and implements functionality from up to version C++11. CMake is an open source make system that uses platform-independent configuration files to generate workspace files for different IDE's and compilers [3]. The developer needs to know or learn how to set up a project in CMake since the engine is platform independent. JavaScript Object Notation (JSON) is a lightweight data-interchange format [4]. The developer needs to know this language to be able to create a set of actors and place them in a world.

With this as a basis, it will be relatively simple to set up a basic scene with models and light. The developer will not need to study low level topics as scene graphs, rendering and matrix algebra to be able to use the engine, as all this will be covered by the API. The developer shall receive relevant errors and warnings in the command window when developing on our engine.

2.3 Reliability

The engine will run stable without stuttering. It will also be robust so the developer do not accidentally crash it and will make sure no unhandled exceptions can occur from the engine's side. There will not be any significant drops in performance during heavy load and memory usage needs to be handled properly when it comes to the time manipulation module.

2.4 Performance

The engine shall manage to keep stable at around 60 fps (frames per second). It must be able to have at a minimum 30 animated actors at the same time. It shall be optimised to make scaling and duplication of actors possible without noticeable loss of performance. This has to happen in accordance with the Extreme Programming principles 3.1.3, code optimisation is to happen at the end of the construction phase.

The time manipulation module will use extra memory and it is important that the engine does not use more than 1 MB when logging one actor for one minute. This applies as long as no custom made components are being logged.

2.5 Constraints

The Nox engine only supports 2D and was not developed with a 3D possibility in mind. In the extended engine, the game developer must be able to choose between using 2D and 3D mode.

2.5.1 Time constraints

The project was started in the beginning of January with a final deadline of May 15th, 2015. Due to the short project period of four months, time was the main concern through the whole project. The time that was assigned had to suffice for the modules planned. The milestones that can be read about in Appendix G, were set to alleviate this problem.

A project plan had to be delivered to our supervisor before the development started. This was to give Gjøvik University College an initial overview of the project and the team members. By the end of February, a website had to be created so interested parties could follow the work.

2.5.2 Software constraints

Nox

The system had to be designed in a way that integrated easily into Nox and followed the design of Nox as close as possible to best avoid having two completely separated designs in one engine. This meant that the modules could not be freely designed. However, the extension to the Nox engine had to avoid breaking its support for 2D. The 2D performance had to remain the same after the Noxplus was integrated.

As mentioned in Section 3.4.1, the coding standards and conventions that Suttung has created for writing readable code had to be followed. The commenting was done so documentation could be generated by Doxygen. This had to be in English, just as the code, to support a wider audience.

2.5.3 Expandability

Since the Nox engine and the Noxplus extension is open source it was important that the engine remained expandable. When the work is published, other developers will expand on the system, change modules and improve it. This made it important that the modules created are easily understandable, removable and reusable. The third party libraries will be possible to replace without effecting the rest of the engine. Other developers who use the engine will also have to follow this principle.

Third-party libraries

The model loader brought some limitations to what types of model files the engine can load. It was possible to write custom file loaders, but due to time 2.5.1, this was not an option. Other third-party libraries had to be adopted in a way that they were well integrated but could be easily changed without destroying the engine as a whole. Thus, none of the components were to be fully dependent on a particular third party library. The physics engine used had to be fully deterministic for re-simulation for the time manipulation module. Read more about it in Section 4.3.5.

2.5.4 Interoperability

One essential part of the Nox engine is that it is cross platform. One requirement from Suttung was that this was maintained. As new modules were implemented the engine had to be tested regularly and made sure it was still working on Linux, Windows and Mac, read more about it in Section 6.3. It was also important that the third-party libraries that were used also ran on the different platforms.

2.5.5 Hardware constraints

The computer that is to be used for development with this engine has to have a graphics card that support OpenGL version 3 or above. It also needs at least 1MB per actor of free RAM per minute when using the time manipulation module.

2.6 User documentation and help system

The demo was created to show the potential and capability of Noxplus, in addition it can be used by developers to see how things work with the new extension. This should work well in collaboration with the demos Suttung created for the Nox engine.

When Suttung published the Nox engine, a Google Group [5] was created to provide support for game developers, contributors, the Noxplus team, and other interested parties. The same group will be used for support on the Noxplus extension.

2.7 Licensing, laws and regulations

It was important that the project did not conflict with the project agreement or the MIT license for the Nox engine. When using resources from other designers and developers the license of these had to be followed. This included all API's, third-party libraries, code libraries, textures and other assets that are not already a part of the original engine. If no licence was found credits had to be given to the author, both in the code and in the credits in this thesis. Standard code snippets found on any programming discussion forums or web pages was not required to be credited as long as parts of it were rewritten. For all tools that are used, the accompanying laws and licences must be followed.

The final product will have the MIT license, this to minimise any trouble for Suttung. This makes it possible for other developers to use only certain parts of the engine without having to comply to different licenses. Because of this, it was important the code is readable and flexible so that Suttung, game developers, enthusiasts, engine developers and contributors understands the code.

2.8 Testing

Source code will always be tested by another group member before it is pushed to the stable branch. Before large changes to the work a push to the stable branch will be performed. Informal performance/stress testing will be done regularly to detect bottlenecks early. Read more in Section 3.3 GoogleTest is as C++ test framework that has support for multiplatforms []. Testing will be done with GoogleTest. Read more about what was tested in Chapter 6.

2.9 Deployment

On the deadline at May 15th, a pull request containing the new features from Noxplus was created to the Nox engine. Suttung will handle it from that point.

3 Development Process

3.1 Project workflow

Documents that were created for the project, both for Gjøvik University College and the ones based on artifacts from RUP, were incorporated into this thesis. They are therefore not added as an appendix. The following Section is an overview of those artifacts. The artifacts have been modified to fit the project's needs better. Some of the topics written in the document for Gjøvik University College covers artifacts from RUP, in that case these were added to the following overview. You can read more about the choice of development methodology in Section 1.4.1.

3.1.1 RUP

RUP provided us with requirements, analysis and design in the inception and elaboration phase [1] [6]. Figure 2 summarise what artifacts from RUP that were used. These are further described later in this Section. The construction phase was substituted with Scrum and some principles of Extreme Programming, as described in Section 1.4.1.

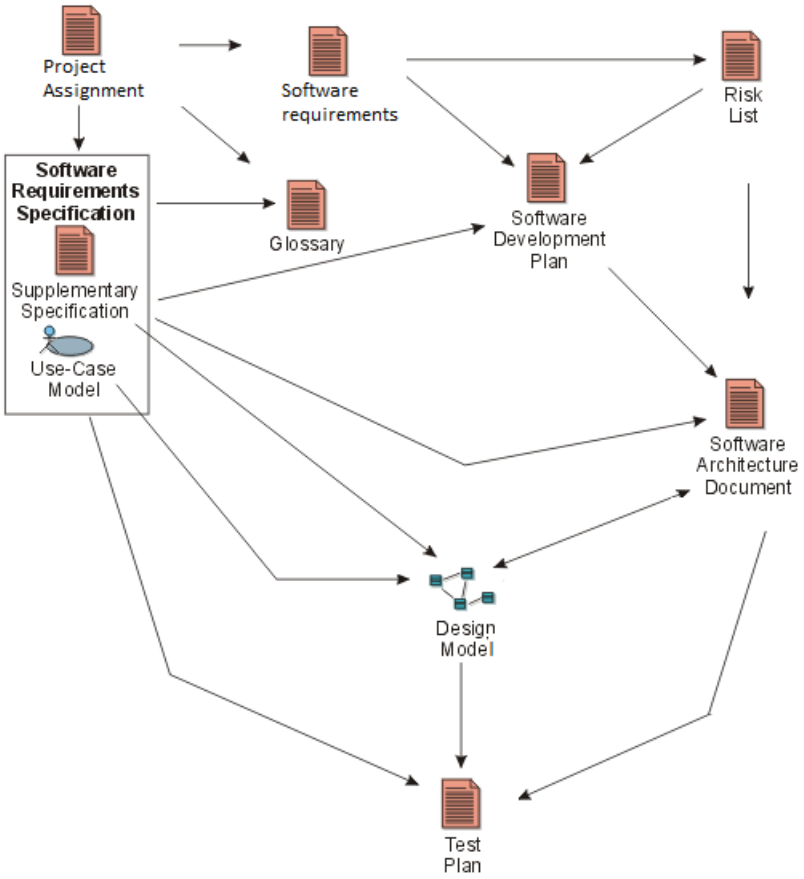


Figure 2: The different RUP artifacts/documents that were used.

Inception:

During the Inception phase the project and its delimitation's were established. The following artifacts from RUP were chosen, and was the focus during the inception phase:

- Risk list
  The risk list was intended to help reduce the possibilities of potential dangers and make the team more prepared if problems occurred. This was updated throughout the project, read more in Section 1.3.4.
- Project glossary
  To make the thesis easier to read, a glossary was included for the project as a quick look-up for abbreviations and words. Instead of having three different ones as RUP suggest, this artifact is modified to only use one glossary where words, expressions and abbreviations that are used throughout the thesis are listed. This is located in Section 1.5.
- Software development plan
  This document encompasses the chosen artifacts and what parts that fit for Noxplus:
  - Risk Management Plan. The risk management plan was modified and combined with the risk list, setting an extra column containing strategies to avoid or eliminate the threats. A table showing how we rate the different risks with regards to likelihood and impact 1.3.4 was also created.
  - Programming guidelines.
    The programming guidelines were the ones Suttung set up as mentioned in 3.4.1.

  This was all the artifacts that were used from the Software development plan. The 14 others are for bigger teams and larger project and they did not suit this project.
- Requirements
  - Software requirements specification
    The use case model can be found under 4.1, with different types of use case descriptions following that.

Elaboration:

In this phase a detailed project plan was created where the requirements and system architecture were specified.

- Requirements:
  - Supplementary requirements
    This is the requirements for the extension when it comes to platform, legal and regulatory requirements, application standards, quality attributes, including usability, reliability, performance, supportability requirements and design constraints. All of this is written in Chapter 2.
- Analysis and design:
  - Design model
    This was used to describe the classes and the relationship between them, it serves as an abstraction of the system giving the team a better overview on

how the classes interacts with each other.

- Software Architecture Document This document covers aspects of the Design. An overview of the architecture were created based on the Design Model and use cases, where the outline of the classes within different layers are drawn up and where the communication happens between them. This helped the team develop the system better, giving an idea how the classes and different layers come together to form the extension as a whole.
- Revised risk list An updated risk list were made in the Inception phase. This lists the risks with solutions or the prevention of them, see Section 1.3.4.

Deployment:

See Section 2.9.

3.1.2 Scrum

Scrum was the choice of methodology for the construction phase. Each day started with a daily scrum meeting using the voice over IP client, Skype. Due to the fact that no regular working hours were set, the daily scrum meetings started once every team member was ready and logged onto Skype that day. The meetings ran for about 10 to 15 minutes where the team discussed what was done the previous day and what was to be done that day. It was helpful to leave the Skype call on after the meetings as a channel for discussing problems and solutions while working.

The team decided to use one week sprints which started each Wednesday and lasted until Tuesday. During a sprint period, a set of tasks were allocated to be done before the sprint's end. The main reason for having one week sprints was because there were regular meetings with supervisor and Suttung on the same day as the sprint started. By ending each sprint a day before the meetings, the team could prepare and plan a new sprint and have it reviewed by supervisor and Suttung on the meetings the next day. After each sprint ended there was a short sprint review and retrospective meeting trough Skype where the team discussed what was accomplished during the sprint, what was done right and what could be done better in the upcoming sprints J. Read more about our sprints in 3.3.

From Scrum, the following artifacts were selected:

- Product backlog gave the team a good prioritised overview of the tasks to be completed.
- Sprint planning meeting helped the team decide on which tasks to be implemented in the upcoming sprint.
- Sprint backlog gave the team a structured overview of the tasks to be completed in the ongoing sprint.
- Daily Scrum meeting increased the overview of the sprint progression and kept everyone updated on the current status.
- Sprint review gave the team the opportunity to tell each other what had been accomplished during the sprint.
- Sprint retrospective meeting gave the team the opportunity to reflect on what was done wrong and what was good.

JIRA was used for keeping the backlog and sprints. More about the sprints and use of JIRA is described in Section 3.2.2.

3.1.3 XP

From Extreme Programming the following principles were used [7]:

- Code refactoring and design improvements were done continuously through the whole project, but design changes were discussed with the rest of the group first.
- Team members followed the same coding standard.
- Collective code ownership gave all team members a better overview of the project.
- System metaphor made the code more readable for other developers and the team.
- Code optimisation at the end of the construction phase, so that all functionality was implemented in time.
- Pair programming, to make difficult tasks easier and detect errors earlier in the development phase.

3.2 Project Management

3.2.1 Meetings

On Tuesdays there was a sprint review and planning meeting before the meetings the next day. Every Wednesday at 10:00 a meeting was held with Supervisor Nowostawski, and 11:30 there was a meeting with employer Suttung. There was a short log written from every meeting, the meeting log is found in Appendix H.

In the daily meetings that were held every day, a review was done on what each group member had done the previous day and what was to be done the current day. Any minor development problems were brought up in these meetings so a solution could be found for it. Most of the time this was not needed as all team members were online on Skype with each other most of the day, and was able to solve problems at the same time they arose.

Milestones

After each milestone a meeting was held where the team together wrote a review for the finished milestone. This included a list of what the team had accomplished so far, a list of what was expected for the next milestone and a small section of thoughts for the passed milestone. If a goal in the passed milestone was not completed, the goal's priority was revised and a decision was made if it where to receive a lower priority or be postponed. The milestone review can be found under Appendix G.

3.2.2 Configuration management

Version management

Bitbucket is a cloud based Git solution by Atlassian [8], and was used for managing and sharing the code base of Noxplus. All team members have experience with Bitbucket from earlier projects. The Git repository was mainly maintained using Atlassians GUI-based Git-client, SourceTree [9]. There was both advantages and disadvantages using SourceTree versus the command line. It gave the developers a better overview of the files, branches and commits, but it was not clear exactly what it was doing to the repository behind the GUI. This was a bit frustrating, but it was working with us 95% of the time. Semantic versioning was not used for Noxplus, since Suttung have not done it, the development time is short and the release is handled by Suttung.

Time management

To log the work hours we used Toggl [10], together with the Toggl mobile app. We created our own workspace where all group members were invited. It was a great tool for keeping track of the group and the group members individual working hours. We could see statistics of how many hours we had spent during a sprint, and on which tasks. In addition it gave us statistics for how much we worked in total over the project period. This can be found in Appendix K.

Change management

JIRA is an issue and project tracking software [11]. The issue and project tracking was done using JIRA, which was hosted on Gjøvik University College's servers. The product backlog was kept track of using JIRA, updated before each sprint and changed as development progressed. In the sprint review meetings the sprint was summarised. This involved a review of what had been done during the sprint while comparing and updating the Scrum board in JIRA. The sprint was then closed and the next sprint was created, where we added issues and tasks that had to be done in the coming week. Tasks that was still relevant and had not been closed in the last sprint were either given a lower priority and postponed, or moved over to the new sprint.

Documentation management

ShareLatex is a cloud-based, open source writing tool [12]. This was used to write the final thesis, making it easier to cooperate with the group. The Latex document is compiled and previewed on the go, and everyone can write at the same time.

Google Docs is also a cloud-based writing tool. Logs from our meetings at Gjøvik University College or Mustad were first written on paper and later in Google Docs [13]. All other logs and documentation was written here first. Here all Noxplus team members, Suttung and Supervisor could participate in the writing and had access to the documents at all time.

These tools were used to make sure everyone were up to date and making it easier to cooperate on the documentation part.

3.2.3 Coding environment

Microsoft Visual Studio is an integrated development environment [14]. This was the chosen IDE for the development phase. Suttung helped with testing on Linux and Mac, so there was no need to use any other. To avoid compatibility issues standard C++11 syntax was used.

CMake was used to build project files for different environments, like Visual Studio, and was already used by Suttung. All third-party libraries that were needed also used CMake, so the including of them as sub modules was a more or less automated process once the CMake files were set up correctly. By looking at Suttung's existing CMake files it became easy to get an idea on how to work with it.

3.3 Development workflow

When starting a sprint each group member chose a task they would like to do, and moved that task from "To do" to "In progress" on the Scrum board. Research was then done on the topic to see how such a task could be implemented. In some cases the research had already been done in the elaboration phase, in that case the implementation would start

immediately. If there were multiple ways to do it and the design was not already done in the elaboration phase, a group discussion was initiated on what might be the best choice out from the research that was found. If no consensus was made, a meeting with supervisor and employer was arranged. When a task had been implemented or the task was stable, it was pushed to our main branch and tested by a second group member to make sure that it was stable and did what it was supposed to.

If anyone got stuck or needed help on a programming task, the team members worked together, either two or all three. For the advanced issues this was quite efficient, as many coding errors were spotted immediately by the other developer(s) and feedback could be given instantly. TeamViewer is a tool that lets one or more person(s) view and control another PC from their own [15]. Since all development was done in the team members' own homes, TeamViewer was often used for pair programming.

If there was a large change to the system, the stable code was pushed to a stable branch before it was pushed to the development branch. This was then tested by all group members before being accepted. When a large new task that possibly would break the engine for a longer period of time, a new branch was created where this task was pushed to while the implementation was under way. Doing it this way would let the other developers continue their work without waiting or being interrupted by the larger task. When the task was finished it was tested to make sure that it did what the task said, that it did not break the design of the system or create any performance issues. When it got accepted, the task was moved from "In progress" to "Resolved". Exceptions to this was if a new task was created to resolve a current issue, that did not create any major implication for the rest of the group. Every group member had to agree on such an exception.

When pushing large code modifications to the main branch in git, it would always be done in coordination with the rest of the group. Sometimes there were merge conflicts that needed all developer's attention.

When developing the engine there was used two git repositories, one for the low level development of the engine and one for testing at the game developer level. The first repository was used to host the Noxplus engine, which was a fork of the Nox engine. The second repository was used to host the test project which used the Noxplus engine as a sub module. Whenever a change was made to the engine it was tested using the test project. This way of working made it possible to completely separate the engine from the test code used in the development project. Read about the testing in Chapter 6.

3.3.1 Assets

Only assets that were free to use was used when creating the demo 1.2.2 and doing the performance testing 6.1. This included 3D models and textures. The only requirement was to give credits to the one who created it. This was added in Appendix I.

3.3.2 Tools

The following tools were used during the development:

- Visual Studio 2013 was our chosen IDE, since all team members have experience with it and are programming on Windows.
- Blender, for creating 3D models and exporting them.
- Gimp, for creating textures.
- Microsoft Project, for creating the Gantt chart.

- Microsoft Visio, for flowcharts and diagrams.
- Microsoft Office Excel, for creating performance charts.
- Skype, for meetings and discussion.
- TeamViewer, for doing pair programming.
- Paint, for sketching and discussing ideas.

For management tools see Section 3.2.2.

3.4 Organisation of quality assurance

3.4.1 Documentation, coding conventions and source code

To make sure Noxplus ended up as a quality product that met the plans and fulfilled all the requirements, all work was logged. All team members had their own log where they wrote important choices that were done while developing. This was helpful both during development and during the writing of the thesis.

In addition to logging choices, logging of the working hours was also done, see Section 3.2.2 for more information. It was also important that each group member was followed up on and did testing while in the development phase.

A close working relationship with Suttung was maintained throughout the project, keeping them up to date with what had been done and what was going to be done. They also helped out with testing for Linux and Mac, read more about Testing in Chapter 6.

3.5 Workload

The groups working hours can be found in Appendix K. Each member has worked about 41 hours per week. As seen in the summary from Toggl, some weeks have less working hours than others. This was due to other deadlines each group member had to maintain. The last 6 weeks of the project was most intense. The 35 hours of work per week that was set for each group member in the group rules was met, see Appendix C for the group rules. 600 hours are the expected amount of time a student is to use on the Bachelor assignment and each member managed over >700 hours each.

4 Design

The most comprehensive and critical functionality that was implemented was the 3D support, see Section 2.1. Without this there would be no time manipulation as this depends on 3D. The idea was that the enhanced engine can be used for both 2D and 3D games, and it is up to the game developer to choose one of them. To make it more clearly what is used for the different purposes, all the 3D specific classes have the suffix "3d" to specify that this is a class only to be used when creating a 3D game. The focus was on reusing as many modules as possible from the Nox engine and make a good programming interface that is both easy to understand and easy to use. With a similar structure to the 2D, it is also easier for game developers to switch to 3D after making games with the 2D.

4.1 Use case

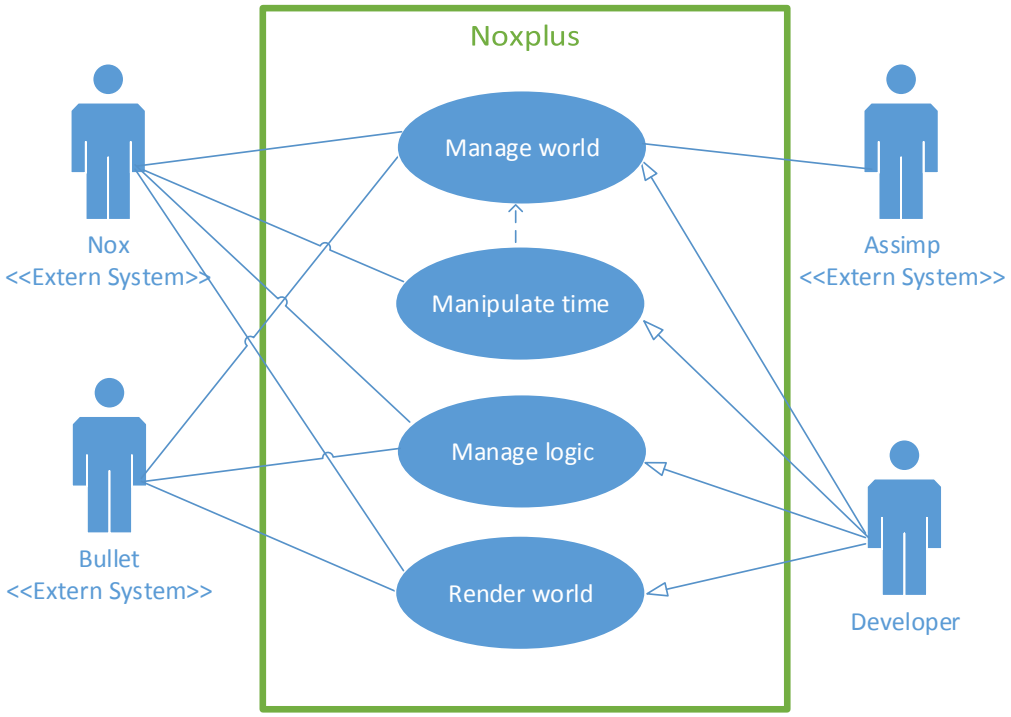


Figure 3: Use case diagram for the new features

The most important developer interactions and external systems for our assignment was found. Use cases was created from them to give an overview of what a developer can use the engine for and where the external systems interact with Noxplus. The model consists of one main user, the Developer, and three external systems, Bullet, Assimp and Nox. Assimp handles the loading of files for 3D models [16], while Bullet is the physics engine that was chosen to be implemented [17]. More about the choice of physics engine and

model loader is written in Section 7.3 and 7.2 respectively. All use cases are developed to work with the Nox engine, making it natural to have Nox as an external system. The use case "Manipulate time" is dependant on that there is a 3D world where it can manipulate the time. The Developer is the one who initialises all the use cases.

4.1.1 Risk analysis of use-case

The risk analysis of the use cases were created to uncover what elements in the planned extension that was the most critical. The risk analysis was divided into three main areas. These was:

- Technological risks: How likely it was that the team did not have adequate technology/knowledge to perform/implement the different functions.
- Project risks: To what extent the different functions could lead to problems in relation to cooperation and organisation in the system.
- Employer risks: To what extent the different functions was essential for the extension to satisfy the requirements.

Use case	Technological	Project	Employer	Total
Manipulate time	High	High	High	High
Manage logic	Medium	High	High	High
Render world	Low	High	High	Medium
Manage world	Low	Medium	High	Medium

Table 2: Illustrates the risks for our use-cases in three main areas.

As the table shows the most risky use case for the system is the "Manipulate time" use case. This is because the area of time manipulation is a little explored area within games and there is little information on this area. This is also the most important module for the extension.

Another high-risk use case of the project is the "Manage logic". The team have never worked with any physics engine before, and Noxplus is fully dependent on physics in our world. A solution could have been to write our own physics engine, but functionality such as complex collision shapes, collision detection and ray tracing that are being used, are highly advanced areas and would have taken considerable amounts of time.

The rest of the use cases were considered of medium risk to the extension of the Nox engine. These were not as technologically advanced as the previously mentioned ones.

4.1.2 High-level use-case description

Use case	Manipulate time
Actor	Game developer and Nox
Purpose	Game developer can manipulate time in game.
Description	The developer can turn this case on or off. Time can be played backwards stopped and played forward again. The developer also have the possibility to set three custom made function. These are run at different times; When the playback is stopped, when the playback is finished and when the playback is started. The Nox engine's event system manage rendering of new actors and avoids rendering of actors that are actively being hidden during rewind and replay. It is also used to detect changes in the world so they can be logged. The data of the objects are only logged when they have been changed.

Use case	Manage logic
Actor	Game developer, Nox and Bullet
Purpose	Manage physics, actors and handle event
Description	The developer can change the properties for the physical world and control the physics such as position, speed and torque of each actor individually. Bullet will respond by triggering events when actors move or collide. An actor component for the physics is used as the API the Developer can use to change physic parameters. The event system of Nox is used for collision and among others logging of the world. The stepping time of the physics is decided by Nox's timer.

Use case	Render world
Actor	Game developer, Nox and Bullet
Purpose	Trigger drawing of the world
Description	The developer decides when the window should draw an image, and Bullet controls drawing of its own debug data visualising the collision shapes. Utility functions from Nox is used for loading and compiling shaders.

Use case	Manage world
Actor	Game developer, Nox, Assimp and Bullet
Purpose	Add and remove actors from the world and manage all actor components.
Description	The developer is able to add and remove actors from the world and find, add, remove and modify all actor components. Bullet and Assimp are only able to modify their own respective component. The Nox engine's world manager is used for managing all the actors.

4.1.3 Expanded use-case description

The two most risky use cases was turned into expanded use cases, because of the findings in the risk analysis 4.1.1.

Use case	Manipulate time
Actor	Game developer and Nox
Goal	Game developer can manipulate time in in-game
Description	The developer can turn this case on or off. Time can be played backwards stopped and played forward again. The developer also have the possibility to set three custom made function. These are run at different times; When the playback is stopped, when the playback is finished and when the playback is started. The Nox engine's event system manage rendering of new actors and avoids rendering of actors that are actively being hidden during rewind and replay. It is also used to detect changes in the world so they can be logged. The data of the objects are only logged when they have been changed.
Pre-Conditions	Time manipulation is enabled.
Post-Conditions	N/A
Special Requirements	For logging and rewinding custom-made components, the developer must implement custom "onSave" and "onRestore" functions for those components that is supposed to be logged. The function "logComponentData" must be called each time the component is to be logged.

Detailed course of events:

Game developer action:	Engine response:
1. The use case begins when the developer starts the engine.	
	2. Starts logging.
3. Starts rewinding.	
	4. Starts rewinding what it logged.
5. Stops rewind and starts replay.	
	6. Runs the function the developer has set. It then starts replaying.
7. Awaits playback to finish.	
	8. Playback finishes and it runs the function set by the developer.

Alternative Scenarios:

1. The developer did not initialise the module. No time manipulation is possible.

6. The developer has not set any function. Engine will just replay.

6. The developer has set a function that spawns or removes an actor, event is sent.

7. The developer stops the playback. Engine runs the function the developer set for this, if any.

7. Event is sent if the developer removes or spawns an actor in the function.

8. The developer has not set any function. Engine will keep running and logg as normal.

8. Event is sent if the function spawn or removes an actor.

Use case	Manage logic
Actor	Game developer, Nox and Bullet
Goal	Manage physics, actors and handle events
Description	The developer can change the properties for the physical world and control the physics such as position, speed and torque of each actor individually. Bullet will respond by triggering events when actors move or collide. An actor component for the physics is used as the API the Developer can use to change physic parameters. The event system of Nox is used for collision and among others logging of the world. The stepping time of the physics is decided by Nox's timer.
Pre-Conditions	1. Logic and physics are created.
Post-Conditions	N/A
Special Requirements	N/A

Detailed course of events:	
Game developer action:	Engine response:
1. The use case begins when the logic is started.	
	2. Engine starts updating the logic and the physics.
3. Developer creates an actor with physical properties.	
	4. Engine creates physics body for the actor.
5. Developer change the actors physical properties.	
	6. Engine applies the physical changes for the actor.
	7. Physics is updated. Positional changes that happens to a physics body are applied to the related transform component.
Alternative Scenarios:	
4. If developer provides conflicting physical properties, e.g giving a static object a mass property, the conflicted property will not be used.	
7. If collisions between actors are detected it will run the collision functions set by the developer or broadcast the collision event.	

4.2 Program flow

The program starts when the application enters the main execution loop managing user inputs and system events before updating all the application processes.

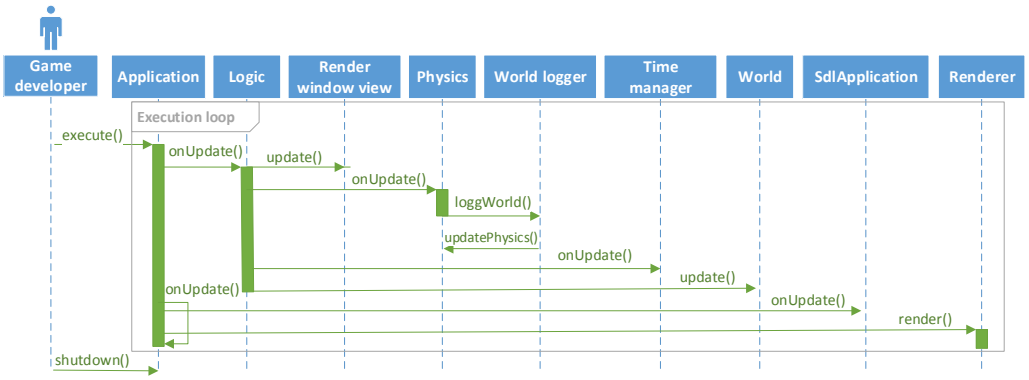


Figure 4: System sequence diagram

When the execution loop is started the application calls the onUpdate function for all its processes; Render window view, Physics and Time manager. The Render window view updates the view which is used to render the logic. The Physics updates all the actors' physical states in the game's world and tells the World logger to log each actor that changes. If the world logger have previously logged data for the current frame it will update the physics using the stored data. The Time manager will update the physics world accordingly to the current mode; regular play, replay and rewind. After Application is done updating its process it will update itself by calling the onUpdate function which the game developer implements. The game developer calls the render function here if she wants to render the world.

4.3 Modules and submodules

A component is a piece of functionality for an actor. See the Nox thesis for full description [18].

4.3.1 Physics

The `ActorPhysics3D` is the component for an actor that is to be affected by physics. When an actor is created from the JSON file it is possible to create this component and enter what physical values the actor is to have. If a value for a physical attribute is not set in the JSON file a default value is given. The values that are available are dependent on the collision shape that is set in the file. The game developer can choose what kind of shape she would like, it can be simple shapes like a box, sphere or cone. It can also be a more complex shape like one assembled by many basic shapes (compound shape) or one that is based upon the mesh of the actor's 3D model (concave or convex shape). There are a couple of shapes that can only be static; like the plane and the concave shape. If no collision shape is given to the actor it will not have the possibility to have physical properties and is not able to detect collisions.

```
"goblinSoldier":
{
    "components":
    {
        "Physics3d":
        {
            "type": "dynamic",
            "shape": "convexhull",
            "size": { "x": 3, "y": 3, "z": 6},
            "quick_cd": true,
            "mass": 10,
            "angularFactor": 0.0,
            "angularDamping": 0.0,
            "linearDamping": 0
        }
        ...
    }
    ...
}
```

Figure 5: Example of a physics component entry in the JSON file.

The Figure 5 shows how a physics component can be set up. Here the actor `goblinSoldier` is defined as a dynamic actor and it uses the models mesh as the collision shape. Using the mesh as a collision shape is quite expensive for calculations and collision detection, but this is avoided by using the `quick_cd` parameter. This parameter tells Bullet that when creating the shape it can optimise the shape by removing parts of the mesh, this is recommended to avoid heavy computations. See performance difference in Section 6.1.2. The `mass` needs to be above 0 or else the actor will not move.

```
void applyCentralForce(const glm::vec3& force);
void applyCentralImpulse(const glm::vec3& impulse);
void applyImpulse(const glm::vec3& impulse, glm::vec3& relativePosition
    );
void applyDamping(float timeStep);
void applyGravity();
void applyTorqueImpulse(const glm::vec3& torque);
void applyForce(glm::vec3& force, glm::vec3& relPos);
void applyTorque(glm::vec3 torque);
void setLinearVelocity(glm::vec3 velocity);
void setAngularVelocity(glm::vec3 velocity);
void setTransform(glm::vec3 position, glm::quat rotation, glm::vec3
    scale);
void setAngularFactor(float factor);
void setSleepingThresholds(glm::vec2 angAndLin);
void setAnisotropicFriction(const glm::vec3& anisotropicFriction);
void setCenterOfMassTransform(const btTransform& xform);
void setCollisionFlags(int flags);
void setDamping(float linearDamping, float angularDamping);
void setFriction(float friction);
void setGravity(const glm::vec3& acceleration);
void setIgnoreCollisionCheck(bool ignore);
void setInterpolationAngularVelocity(glm::vec3& angularVelocity);
void setInterpolationLinearVelocity(glm::vec3& linearVelocity);
void setInterpolationWorldTransform(btTransform& transform);
void setLinearFactor(glm::vec3& linearFactor);
void setRollingFriction(float friction);
void setRestitution(float restitution);
```

Figure 6: All the apply and set functions for an actors physics component.

In addition the developer can use the physics component to give an Actor new physical properties or retrieve them through many different get and set functions. Figure 6 shows all the set and apply functions available for a game developer. There are equivalent get functions for all the set functions.

Basic shapes

There are 6 basic shapes that can be used for collision detections, these are the fastest for computing collision.

- Box, needs the size as a vector.
- Sphere, is in need of the radius as a float.
- Cylinder, requires the size as a vector.
- Capsule, is needing the height as a float and the circle radius as a vector.
- Cone, needs a circle radius as a vector and the height as a float.
- Plane, can only be static and needs a plane offset as a vector and a constant as a float.

Advanced shapes

The more complex ones are needing a bit more information.

- Concave hull shape, can only be static and requires a mesh, this is used to create the shape.
- Convex hull shape, requires a mesh, this is used to create the shape. Can also pass

in a scaling scalar to scale the mesh shape. In addition a Boolean `quick_cd` can be passed, to say if the mesh is to be optimised so that it takes less computational power, this is recommended.
- Compound shape, requires at least two shapes and their data.

Collision callbacks

The game developer has the possibility to decide what should happen to an actor when it collides. This can be done in two ways. One option is to write a function and register it in the simulation class together with an actor ID. The function parameters gives the game developer access to both actors that collided, and the game developer can decide what to do with both. This is done through an `std::function`. Another option is to let the game application listen for collision events and parse each of them when received. The event that is received contains the collision information on what actors collided, together with where the collision happened on the actors and the forces. The game developer can choose what to do with this event.

4.3.2 Assets

More than 40 different file formats are supported by Assimp. The following file types have been tested with good results: ".obj", ".dae", ".md5". For model files with more than one animation, using the ".md5" file format is recommended. When there is only one animation ".dae" can be used. The ".obj" file does not support animation and can therefore only be used for loading static models.

Loading models

Actors that has a 3D model needs a path in the actor's JSON file that tells where the model's file is located. Together with the file path, it also requires a name. See Figure 7 for an example. The `name` is used by the engine to select the right model when the actor is about to be drawn. If multiple actors are using the same `name` under `Graphics3d`, all these actors will be drawn with the same model. The reason it is done this way is optimisation; say the game developer wants to spawn 100 trees using the same actor file. All the tree-actors will have the same `Graphics3d` name. The model will be loaded one time when the first actor is created, and all the other 99 will be set to use this model. For the textured models, the file paths of where the texture files are located are stored in the model file and the textures are loaded automatically. There is a limit to how many actors that can live in a scene, see Section 6.1 for more information.

Animations

If the model file contains animation data, a `startAnimation` index can be specified in the JSON file. The indexing starts at 0 and goes up to the number of animations -1. The value tells which animation that should be played when the game starts. If no value is set, or if the index is invalid, no initial animation will be played for the actor. The `animationSpeed` sets how fast the animation should be played. Value 1 will play the animation at its normal speed, 2 will double the speed, 0 will pause it, and negative values will play it backwards. `animationStartTime` can be set to specify how many seconds into the animation it should start playing at. By default it will automatically start playing at the beginning of the animation.

Information about the active animation is stored in the actor's `ActorGraphics3d` component. The animation can be changed during runtime with the following function:

`ActorGraphics3d::setAnimation(int index, bool logIt, float speed, float startTime)`.

If the animation change should be logged by the world logger, the `logIt` argument must be true. This will make the change of animation visible during rewind and replay.

```
"goblinSoldier":
{
    "components":
    {
        "Graphics3d":
        {
            "name": "goblin",
            "dataPath": "assets/models/goblin/Goblin.dae",
            "startAnimation": 0,
            "animationSpeed": 1.0,
            "animationStartTime": 10.0
        }
        ...
    }
    ...
}
```

Figure 7: Example of a Graphics3d entry in the JSON file. The actor is using an animated 3D model.

4.3.3 Rendering

Like with the Nox engine, for rendering, the game developer should create a custom window class that inherits from the `RenderSdlWindowView` class. The constructor of `RenderSdlWindowView` takes two Boolean arguments; `create2dRenderer` and `create3dRenderer`. It is possible to set both to true. In that case, it will create one 2D and one 3D renderer. The 2D renderer will use Nox's OpenGlRenderer class, while the 3D renderer uses the Noxplus' OpenGlRenderer3d class. These flags will only control the rendering; to render both 2D and 3D, a 2D logic must also be created first.

Noxplus comes with a generic camera. An instance of the camera must be created by the game developer, and set in the renderer. The only value that is passed into the constructor of the camera is the size of the screen. To change it, the values in the constructor must be changed.

An object of the window should be created in the user created `Application` class 4.3.6. The window's `render()` member function should be called in the application's `onUpdate()` which is run about 60 times per second. Animations are updated by the engine according to the active play mode; pause, rewind, replay or normal.

Light

```
"Light3d":
{
    "lights":
    {
        "light name":
        {
            "type": "directional/spot/point",
            "offsetPosition": {"x": 0.0, "y": 0.0, "z": 0.0},
            "cutOfAngle": 20.0,
            "direction": {"x": 0.0, "y": 0.0, "z": 1.0},
            "color": {"r": 1.0, "g": 1.0, "b": 1.0},
            "range": 10.0,
            "ambientIntensity": 1.0,
            "diffuseIntensity": 1.0,
            "constantAttenuation": 1.0,
            "linearAttenuation": 1.0,
            "exponentialAttenuation": 1.0,
            "rotateWithActor": true,
            "castShadows": false
        },
        "light name two":
        {
            // Light data
        },
        ...
    }
}
```

Figure 8: Example of a Light3d entry in the JSON file.

The game developer can create and attach multiple light sources to an actor using the `Light3d` component, this is done by specifying a `Light3d` entry inside the actors JSON file, under the components section. A typical entry for this component is shown above in Figure 8, where the different properties describes:

- Light name is used as an index when referring to the light.
- Type is either directional, point or spot light.
- OffsetPosition is the position used to offset the light from the actor position.
- CutOfAngle is the cone angle in radians.
- Direction describe the lights direction.
- Color is the light colour represented in RGB format.
- Range describes how far the light shines.
- AmbientIntensity describes the intensity of the ambient lighting.
- DiffuseIntensity describes the intensity of the diffuse lighting.
- ConstantAttenuation describes the constant light strength reduction.
- LinearAttenuation describes the linear light strength reduction.
- ExponentialAttenuation describes the exponential light strength reduction.
- RotateWithActor is used to decide if the light should rotate with the actor or not.
- CastShadows is used to decide if the light should cast shadows or not.

The game developer can create multiple lights inside the `lights` section. Each light entry inside the `lights` section is started with the `light name` followed by a set of properties describing the light to be created. Directional, point and spot light have a few common required properties:

- OffsetPosition.
- Color.
- AmbientIntensity.
- DiffuseIntensity.
- RotateWithActor.
- CastShadows.

In addition to these common properties each of the light types requires additional properties. Directional light needs `direction`. Point light needs `constantAttenuation`, `linearAttenuation` and `exponentialAttenuation`. Spot light needs every one of them.

4.3.4 Actor control

```
"vectorControls":
{
    "actions":
    [
        "move",
        "rotate"
    ],
    "buttons":
    {
        "W":
        [
            {
                "action": "move",
                "vector": {"x": 1.0, "y": 0.0, "z": 0.0}
            }
        ],
        "E":
        [
            {
                "action": "rotate",
                "vector": {"x": 0.0, "y": -1.0, "z": 0.0}
            }
        ]
    }
}
```

Figure 9: Example of mapping controls with a JSON file, using directional and rotational control component.

The game developers can handle basic control of the actors using the `Directional-Control3d` and `RotationalControl3d` component. To use these components the developer needs to create a JSON control file which maps action events to different keys, see Appendix L for a full overview of the different control types that can be created. Both control components are using vector control, meaning that the control data is rep-

resented as a vector of 3 where the values range from 0 to 1. The right hand coordinate system is used. In Figure 9 there is shown an example of a JSON control file creating actions for the directional and rotational control and mapping them to two different keys. The actions section is where the actions are created, the move action is used by the directional control and the rotate action is used by the rotational control. The actions can be mapped to different keys in the buttons section. For example the W key is mapped to the action move. In this example the vector is x = 1, y = 0 and z = 0, this means that the actor is moved to the right every time the W key is pressed.

Directional control

```
"3dDirectionControl":
{
    "movementSpeed": 5.0,
    "relativeToCamera": true,
    "relativeToRotation": false
}
```

Figure 10: Example of a directional control component entry in the JSON file.

The directional control component handles the actors movement in x, y and z axis allowing the actor to move freely in all directions. The movement speed defines how fast the actor will move, in meters per second. The `relativeToCamera` and `relativeToRotation` describes if the controls should be relative to the camera or the actor rotation or both. For example if "relative to camera" is enabled, the left, right, up and down directions will be transformed accordingly to the cameras coordinate system. The "relative to actor" works the same way using the the actor coordinate system. If it is not relative to the camera, nor the actor, it will be relative to the OpenGL right handed coordinate system.

Rotational control

```
"3dRotationControl":
{
    "rotationSpeed": 5
}
```

Figure 11: Example of a rotational control component entry in the JSON file.

The rotational control component handles the actors rotational control in x, y and z axis. The `rotationSpeed` is a scalar describing how fast the actor is rotated using radians per second.

4.3.5 Time manipulation

Mapping of time manipulation

Before starting with the time manipulation module a plan had to be made on what kind of time manipulation that was to be implemented. Scenarios for different time travelling theories was discussed and mapped, both possible and not possible ones for game engine implementation.

The Multiverse theory expands on the theory of Big Bang. When the Big Bang hap-

pened it did not just happen once but an infinite number of times. This is the inflationary variant of the theory and there are many others [19] [20].

Multiverse	
Scenario	**Example**
Branching timeline.	If a change is made after travelling back a new timeline branch is created. You can now travel between them, and play forward again. Time flows normal in the timeline you are not in even when rewinding or replaying in the other timeline.

Table 3: Scenario for the Multiverse theory.

Dynamic timeline	
Scenario	**Example**
Rewind time, change something, replay and see a difference.	The comic "The Order of the Stick" has an example; murdering a person in the past, kills the whole family tree from the dead person's place in the tree, back in the future [21].
Rewind time. If a change is made after travelling back in time the future will be discarded.	This is done in the game "Braid" where the player can rewind back in time. If he makes a change, he cannot replay back to the feature [22].
Travel back and forth between snapshots of the world. If a change is made after travelling back to a snapshot the future snapshots will be updated.	The game "Dark Chronicle" lets you make changes in the past and the future is updated [23].

Table 4: Scenarios for the Dynamic Timeline theory.

Table 4 shows the possible scenarios we came up for where the action in the past is affecting the future [24] [25].

Fixed timeline	
Scenario	**Example**
Pause world, give commands and then resume the world.	The game, "Pillars of Eternity", lets you pause the time whenever you want in combat so that you can give your team commands before continuing. [26]
Pause world, walk around in the paused world and do stuff before pressing play again.	This is done in the movie "Click" where the main character pauses the time to punch an annoying boss in the face. When the time is resumed the boss was in pain [27].
Speed up/slow down world or selected actors while playing.	This is done in the movie "Click" where the main character slows down or fast forward the time when something interesting or uninteresting happens [27]. Receive other people's time, they lose the time they gave away, while the other gain it. Increase speed of time for the person who gave away time and the other who gained will slow down their speed. World moves at normal.
Rewind without making any changes.	This is done in the movie "Click" where the main character travels back in time in order to gather information about how he and his wife first meet [27].

Table 5: Scenarios for the Fixed Timeline theory.

The Table 5 is a collection of scenarios we came up with for the fixed timeline theory or closed timeline curve. Meaning that if a change to an object occurs in the past it will return to its original state in the future [28] [29].

Fixed or Dynamic timeline	
Scenario	**Example**
Rewind time to interact with your past self.	"Company of Myself" is a game that uses this kind of mechanic [30].
Travel back and forth between snapshots of the world and time.	A player takes a snapshot of the world, keeps playing. After a while he takes a new snapshot. Player can now travel between these two points.
Travel between different eras.	In the TV programme "Doctor Who" there are both fixed points in time that cannot be changed, but the rest can.

Table 6: Scenarios for either Fixed or Dynamic Timeline theory.

Certain scenarios fits under both dynamic and fixed timeline, depending on implementation. Another Table was created 6.

The group decided that with careful planning and given the short amount of time, the following scenarios would be possible to implement in the game engine:

- Rewind without making any changes.
- Rewind time to interact with yourself.
- Rewind time, change something, replay and see a difference.

Choosing implementation

There was multiple ways of how the rewind and replay functionality could be implemented. A good source of inspiration came from the game "Braid" [22] and a video where Jonathan Blow, the developer of Braid, talks about how he implemented the rewinding [31]. The team created a list of different options:

- Store the entire world each frame when playing, and then rewind and replay using the stored world.
- Store the world changes at intervals and interpolate between the intervals.
- Store the world changes at intervals and re-simulate between them.
- Store snapshots of the world at intervals and store just the changes between them.
- Store physics events and re-simulate them after rewinding.

Re-simulating the physics backwards would be impossible. However, with Bullet being deterministic it should be possible to re-simulate it forward. The problem would be if the simulation of physics was to be done backwards as the rewind technique, without using snapshots of the world where we store the world state. Using snapshots, would give the possibility to re-simulate physics forward, but not backward. When the data has been manipulated to be reversed it could cause deviations and give a different result when rewinding. Replaying would work as expected, because the same values are used during the re-simulation. It would not be possible to rewind using negative delta time since Bullet does not support it. When negative delta time was passed to Bullet it seemed like the time stopped and boxes floated around, you can see a picture of it in Figure 12. However the world does run and you can move the camera around and spawn actors, they will also float around.

Figure 12: Stepping the physics world using negative delta time in Bullet.

The team decided that the safest way to get a working result in time, was to log all position changes for each actor trough the gameplay. The question was how to make it efficient, and with lowest possible memory usage. To save space, only the actors that change in some way, being position, rotation or animation, was logged. This has been done in other games before, like the 2D platformer Braid [22]. Another idea when it came to efficiency was to store the logged data continuously in memory, with the same order that is was logged. This way the CPU cache would be utilised during lookup of the data later, for example during rewind. Reading data from the memory continuously is faster than looking up fragmented data.

Logging gameplay

The next decision was on how to log the world. One idea was to save the entire world at each frame before the physics were updated. This would take more space, but it would give the developer more opportunities, like swapping directly to different world states. Another idea was to log the same place where the actor's transform component is synchronised with the physics. With the memory usage in mind, it was decided that the best choice would be the last mentioned one. This would make it easy to log just the actors who are updated and avoid logging actors that does not change during a frame.

The next problem was how and when to store data. There were different ideas for this too:

1. Using a map to log physical changes for every actor in the world and use a separate array containing data for each frame, telling which physical changes to use at that frame.
2. Using an array where each entry is a frame. The data of all the actors are stored in

it, if an actor is not updated we will not add it in the entry, since it will not need any changes.

3. Using an array where each entry is a change.

The first option would cause many map look-ups and fragment the data in memory, meaning the extra speed from CPU caching would be lost.

The second option would be problematic. When creating the the array for storing the data, every array element must have enough space to store all the actors that possibly can change. If new actors are spawned during gameplay it might not be enough space in each array element. A solution would be to let the game developer set how many slots there should be at each element, but this is not a memory saving solution.

The third idea was the first prototype that was implemented in Noxplus. This is described in the Section 5.7.5.

Time manipulation API

```
auto worldLogger = make_unique<DefaultWorldLogger3d>(logic);
auto conflictSolver = make_unique<DefaultTimeConflictSolver3d>();
auto timeManager = make_unique<TimeManipulationManager3d>(move(
    worldLogger), move(conflictSolver));

physics->setTimeManager(timeManager.get());
logic->setTimeManager(move(timeManager));
```

Figure 13: Example of how the time manipulation manager is created

When designing the API it was important that it was easy to set up and use for the game developer. To be able to achieve this, most of the complexity of the time manipulation was hidden behind easy to use functions. See Section 5.7 for how it was implemented.

In order to use the time manipulation features mentioned in Section 2.1, that Noxplus provides, the game developer must create a `TimeManipulationManager3d` and add it to the application's logic. All rewind, replay and pausing should be done using the functions provided by `TimeManipulationManager3d`. Time manipulation manager controls the game's time line using three functions; rewind, play and pause. The rewind function plays the time backwards until the play function is called. It will automatically pause the game when the beginning of the gameplay is reached. The play function starts playing the time forward applying the logged data (if any) to the world until the rewind function is called. The pause function stops the time and all the actors in the world freezes in their current state.

In order to create the `TimeManipulationManager3d` the developer needs to create an object of `DefaultWorldLogger3d` and `DefaultTimeConflictSolver3d`. The world logger logs all actor state changes in the world, while the conflict solver is used to solve conflicts or paradoxes that can happen during time manipulation.

Using the conflict solver's `setConflictSolverFunction(..)` the game developer can set a std::function using the actor ID as argument that will be called when a conflict or paradox occurs for that actor. When the function is set, there must also be specified what type of conflicts it is used for. At the time of writing, only the `ACTOR_INTERRUPTED` conflicts are detected. These conflicts occurs when an actor that is being replayed is interrupted, for example if there is a collision between a replaying actor and a

non-replaying actor. Additionally there are three other functions; `setOnReplayStarted`, `setOnReplayStopped` and `setOnReplayFinished`. These are used to set functions that will be run when replay starts, when replay is stopped by the player, and when it is finished replaying the saved data. For example if the game developer is creating a game where the player is interacting with its past self, the game developer might want to spawn a new actor when the replay starts, and remove it when the replay is finished.

Conflict solving

There are numerous possible conflicts that can happened when playing with time. A brain storming was done on what possible conflicts that can occur, and the ideas were categorised and grouped based on which conflicts could be solved using one implementation of a conflict solver. All conflicts that were found had one thing in common; The source of every conflict during time travel is when the past is interrupted from an external source. For example in the grandfather paradox everything starts when a person travels back in time to kill his own grandfather [32]. Did he kill himself? If so how could he travel back in time to carry out the murder if he never existed? This paradox is caused by an interruption done on the grandfather. If the conflict solver could detect this interruption, the game developer would be able to cover most of the conflicts and paradoxes that can occur. The team decided to implement the conflict solver using Bullet's collision callback system that already was integrated in Noxplus, see 5.7.3. If used correctly, the conflict solver can solve a large amount of different conflicts. The game developer can solve the grandfather paradox by attaching an "interrupted"-function to the grandfather which handles the paradox. When the interruption occurs, Bullet will detect it as a collision and the conflict solver will launch the grandfather's "interrupted"-function. What consequences this gets is up to the game developer to decide.

Logging of components

It is possible to make components that can be used with the time manipulation. Components with rewind and replay possibilities need to derive the `TimeLoggingComponent` instead of the normal component class. The `TimeLoggingComponent` class works like a normal component but it provides three extra functions for handling the logging and restoration of component data; `onSave(saveData)`, `onRestore(restoreData)` and `logComponent()`. The first function is used to save component data using the `saveData.save<type>(data)` function. The second function restore the component data using the `restoreData.load<type>(data)` function. The data needs to be loaded in the same order as it was saved, read more about this in Section 5.7.4. The third function tells the world logger to log this component, this function is called by the game developer, for example every time the component data has been changed.

4.3.6 Demo

Suttung have created several demos for Nox that shows examples of how the different features can be used. The Noxplus-team have created one demo that demonstrates and shows the added features of Noxplus , see the demo here [33]. The demo is paused when first launched and is unpaused using the `P` button. Rewind is done by the `Backspace` button while the `Enter` replays. The `middle mouse-button` is used for shooting boxes and the `left mouse-button` rotates the camera. Clicking the `right mouse-button` while hovering over an actor gives control over that actor, and it can be moved by the

`TFGH buttons`. `WASD` is used for moving the camera and holding `shift` while moving increases the speed.

5 Implementation

The Nox-engine is written in C++11. To keep the Nox-engine as consistent as possible the same programming languages and standards have been used for the Noxplus extension. Simple DirectMedia Layer 2.0 (SDL 2.0) is used for access to input devices, audio and graphics via OpenGL [34]. Open Graphics Library 3.0 (OpenGL 3.0) is an API for talking with the GPU [35]. The Nox engine uses SDL 2.0 to manage the graphics window and user inputs, while OpenGL 3.0 is used for drawing the graphics. These are cross platform and can be used both for 3D and 2D graphics. Additionally, Assimp was used for reading of 3D model files. SDL Image 2.0 is an image loading library that requires SDL 2.0 [36]. This was used for loading textures. For simulating 3D physics the open source physics engine, Bullet, was used.

5.1 Logical View

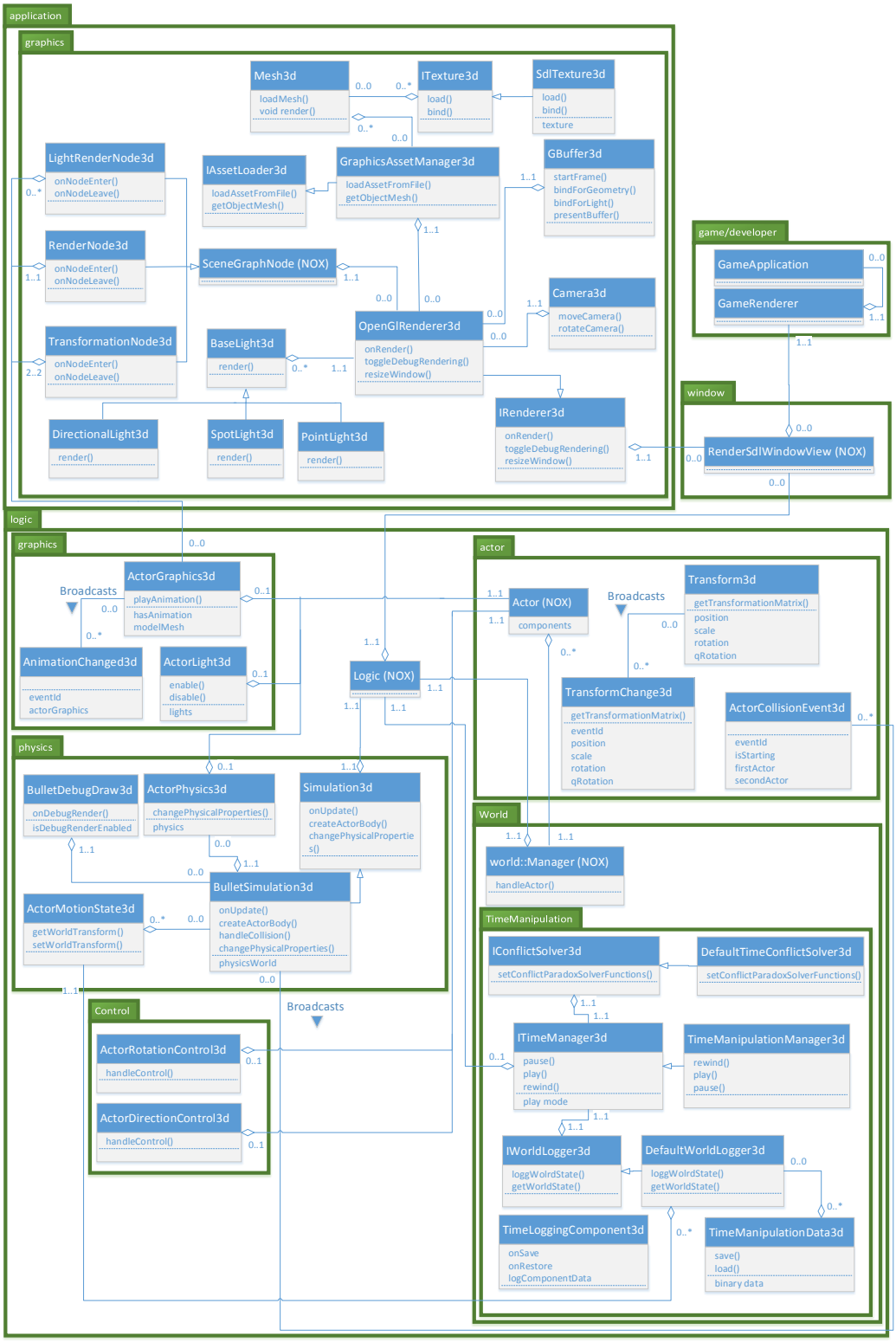


Figure 14: Design Class diagram

The design class diagram in Figure 14 was created from the class diagram 15 and the use case model 4.1, to make it easier to know the structure of the system and how everything connects. The requirements that are set in the Requirements specification chapter 2 were followed when it was created. The design class diagram shows important layers and sub-layers used by Noxplus. Most of the classes are specific for Noxplus except the ones marked "(NOX)". The design follows the layers that Nox use, where there are two major layers. This had to be done to follow the design of the Nox engine as this is one of our requirements 2.5.2. These layers are the application layer and the logic layer. This is the same Design as Suttung mentions in their thesis [18], where they follow the design laid out by the book Game Coding Complete [37]. There are however layers within these two layers to use the famous divide and conquer principle [38], making it easier to develop the system.

The `RenderSdlWindowView`, inside the window layer, is an implementation of the `View` class which is not shown in the diagram. The `View` class is within the logic layer, and the `RenderSdlWindowView` is inheriting from this. It is used by the renderer to "look into" the logic so it knows what to render. Without this it would not find any actors or receive any broadcast messages. It is through this view that the game developer will create his game. Read more about the `View` class in the Pyroeis document [18].

The game/developer layer is not a part of the Nox engine but a separate project depending on the Nox engine for creating a game. The `GameApplication` creates and stores an instance of the window, the logic, the physics, the world, the time manager and the event manager. The window is then added to and managed by the logic as a view.

The logic layer handles everything that has to do with the logic of Noxplus. Inside this layer in the world layer, we created a third layer. This is where the use case `Manipulate Time` was developed 4.1. Not only is the `Manipulate time` layer inside the world layer, but this is also where the `world::Manager` is residing. The developer can get this through the logic context, which again is gained through the `RenderSdlWindow-View`. The Manager class is the most important class for our use-case `Manage World` 4.1.

`OpenGlRenderer3d`, in the application layer, is the main class that keeps track of all the graphics and rendering. It has a connection to the actors through the actor component `ActorGraphics3d` and the render nodes. This is one of two connections between the application layer and the logic layer, the other being the `RenderSdlWindowView`, creating low coupling between the two major layers. This helped with finding faults since the exchange of data is minimal and changing parts of the system became easier since there are just a few classes that needs to be modified. There will also be people who would only want parts of the extension, this way makes it easier to pick it out of the extension.

By dividing the logic layer into smaller ones we get higher cohesion. This gives increased complexity, but a better overview of the system, makes it easier to change modules and the development can happen more in parallel. The increased complexity is something the team can live with since it helps more than it hurts [39].

5.2 System architecture

To get enough time for the time manipulation the 3D module had to be narrowed down. The areas of 3D graphics are endless [40]. The focus was on what the research told us

was the most important.

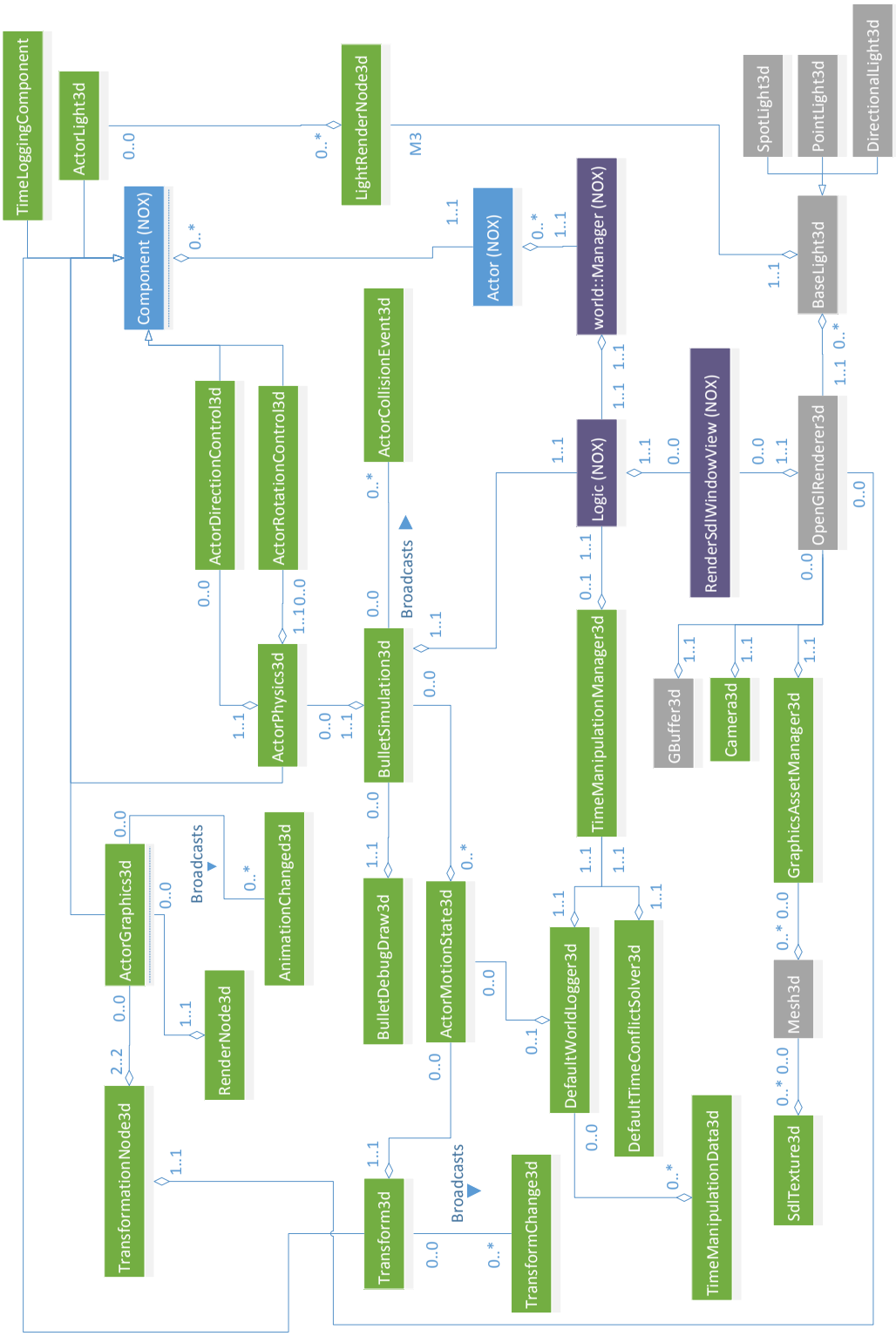


Figure 15: Class diagram

The class diagram shows most of the classes in the system, the only classes who are not shown here are: the interface classes and the 2D specific classes that are not used by the 3D and time manipulation module. It was created out of the use cases and the assignment description and has been reworked multiple times throughout the project. The system has five important classes; `OpenGlRenderer3d`, `BulletSimulation3d`, `ActorGraphics3d`, `Actor` and `Logic`. `OpenGlRenderer3d` handles the rendering of models and lights. It is the main component for the "Render world" use case, since it contains the root node of the scene graph, you can find the use cases under Section 4.1.

`BulletSimulation3d` is the class responsible for physics update and collision handling, and is the class that communicates with Bullet. This is the reason that it has so many connections to other classes. `Actor` is the class that represents an object in the scene; it could be a light or a model. The Actor class has many different types of components that goes through the same state cycles as the Actor. The class was part of the original Nox engine, and only a few parts was modified to fit the Noxplus extension. The Logic class is the class that represents the logic context; it contains the physics, world manager, data storage, broadcaster, time manager and all the Views. It binds the whole logic together and lets other views look into the logic.

Only the most basic fundamentals was implemented first. This was to have something that could run at all times. The team worked in an iterative way 1.4.1, and it was important to always have the possibility to test new features as they were developed.

To differentiate between what has been implemented by the team and what has been reused from other developers open source projects a colour scheme is used. All the green classes is the classes that has been implemented by the team. However there is two exception; Even though the `BulletSimulation3d` is marked green it still contains two functions that have been reused from an external source. It is still being marked green because the reused part is only about 5%. The `SdlTexture3d` class has 10 lines of reused code. The classes marked with grey is the classes that are based on other external open source projects, but have been adapted and integrated to fit the design of Nox. It required a big amount of working hours to analyse and adapt the code to fit Noxplus. The classes marked with purple colour is the the classes that are written by Nox and modified by us. The changes made are minor but it requires deep understanding of the Nox engine and its infrastructure.

5.3 Scene module

5.3.1 Scene graph

The first scene graph implementation was simple. The renderer did only require a render node in the scene graph used for rendering an object. The renderer contained a render node acting as the root node of the scene graph. At this point the renderer was only capable of rendering static objects and all that was needed to be done was to create a new render node and attach it to the root node, whenever a new actor was created. The shaders used to render the objects, only transformed the object to the right world position and gave them a static colour. This was used as a base and later two additional scene graph nodes was added for managing transformation and lighting. Every node overrides two key functions `onNodeEnter` and `onNodeLeave`, these two functions are called recursively when a parent node is traversed. The `onNodeEnter` is called when entering a node and the `onNodeLeave` is called before exiting the node. In addition to these

two key functions, the nodes used for rendering and lighting needs two extra functions, `onAttachToRenderer` and `onDetachedFromRenderer`. The first function is called for the current node and all its child nodes, when a renderer is attached or detached from a particular node. The second function is called when a node is being added, or when it is removed from a node that already has an attached renderer.

Transformation node

The Transformation node handles the transformation of translation, rotation and scaling for its child nodes. This is done in the `onNodeEnter` function which provides a reference to a model matrix which is recursively passed down to the child nodes. Before the reference to the model matrix is being passed down to the child nodes, the transformation node saves the value of current model matrix before transforming it using its own transformation data. The `onNodeLeave` function restores the nodes original state using the previously saved model matrix.

Renderer node

The render node renders an actor using the actor graphics component which contains the actors mesh data, see Section 5.4.2 for detailed explanation. The rendering is done in the `onNodeEnter` function using the uniform handle arguments and a pointer to the actor graphics component. When rendering the actor graphics component it will use the model matrix provided by its parent node. The `onNodeLeave` is left as an empty implementation because for us there is no need for any post render operations. However it could be useful for an engine developer.

Light node

The light render node handles the rendering of a light source attached to an actor. This node do not have the need for the `onNodeEnter` and `onNodeLeave` functions so it is left as empty implementations. This node do not render the lights themselves, but it tells the renderer to store and render the light later. This is done in the `onAttachToRenderer` and `onDetachedFromRenderer` using the renderer pointer argument. The `onAttachToRenderer` will run adding the light to the renderer and the `onDetachedFromRenderer` removes the light from the renderer.

5.3.2 Camera

A basic camera class was created. The initial projection matrix is generated in the constructor. It is used together with the view matrix to update the view projection matrix. The view projection matrix is used in the renderer to generate the model view projection matrix which in turn is passed to the shader to map from object space to world space. Read about the camera controls in Section 5.8.1.

5.3.3 Light

The light is implemented using an abstract class `BaseLight3d` used by `DirectionalLight3d`, `PointLight3d` and `SpotLight3d`. The `BaseLight3d` manages common data and functions used by all the light types. It has three important functions; one `init` and two `render`-functions that the derived classes needs to implement. The `init` function is used to initialise the OpenGL uniforms handles used by the shader programs. As mentioned, there are two overloaded versions of the `render` function. The first function is used to render into the stencil buffer used to delimit the light range, the second

function renders the light.

Point light

The point light needs additional positional and attenuation data to describe the lights position and the gradual loss in intensity, the attenuation is described using constant, linear and exponential attenuation. A sphere mesh is used to render the point light, the radius of the sphere is calculated using the attenuation data making it just big enough to cover the lights affected area avoiding wasted light calculations.

Spot light

Because of the similarity between the point and the spot light, the spot light is derived from the point light. The only difference between the spot and point light is the lights geometry where the point light has the geometric properties of a sphere while the spot light has the geometric properties of a cone. The size of the cone is defined by a cutoff angle which is used to calculate the cone size in the shader.

Directional light

The directional light has an additional direction property which describes the light's shining direction. The light is rendered using a quad mesh scaled after the window size. The directional light affects the whole world so it do not need the render function which renders to the stencil buffer delimiting the light range.

5.3.4 Rendering

Rendering actor graphics

All model rendering is done in the render function of the `Mesh3d` class. During the creation of an actor's `Mesh3d`, see Section 5.5, a new OpenGL vertex array object (VAO) is created and bound to the OpenGL context. This is kept bound while the mesh data of the actor is loaded and buffered to OpenGL, and is again bound later when the object is about to be drawn to the screen.

Some actors have more than one mesh entry in their model file. When the model file is loaded, all vertices, indices, normals, colours and texture coordinates for all the mesh entries will be pushed on their own respective vectors and buffered to OpenGL at the same time. To later render this correctly, which sections of the vectors that belongs to the different mesh entries must be specified. This is done by using the `MeshEntry` struct, in the `Mesh3d` class. For each mesh entry in the model file a `MeshEntry` is created. The struct stores where in the vector the vertices/indices for the particular mesh entry starts and how many vertices/indices it has.

When rendering the `Mesh3d` all mesh entries in the object are looped trough and rendered one by one. The draw call `glDrawElementsBaseVertex` is used, where what parts of the buffered vertex and index data that should be used to draw is specified. Each vertex has a related normal, colour and texture attribute, so the same numbers are used for these. See Figure 16. Without the use of mesh entries, all vertecies would have been drawn as one mesh and there would have been drawn lines between them.

```
glDrawElementsBaseVertex(GL_TRIANGLES,
        entries[i].numIndices,
        GL_UNSIGNED_INT,
        (void*)(sizeof(int) * entries[i].baseIndex),
        entries[i].baseVertex);
```

Figure 16: Render call

Rendering Animations

Skin animated 3D models contains a skeleton of connected bones. For humanoid characters, the bones are often structured like in the real human body, with simplifications. See Figure 22. The skeleton is linked to the animation data created in a 3D modelling program like Blender [41], that tells how the skeleton should be deformed at a given time. Each vertex on the actual 3D model has a maximum of 4 affecting bones with belonging weights, that tells how much impact each surrounding bone has on the vertex. When a bone is moved or rotated, the vertices will move along with different speeds depending on the bone weight for the particular vertex.

For rendering animations, two extra attributes was added to the geometry fragment shader:

```
layout (location = 4) in ivec4 boneIDs;
layout (location = 5) in vec4 weights;
```

and two new uniforms:

```
uniform bool hasAnimation;
uniform mat4 bones[100];
```

The two attributes `boneIDs` and `weights` tells which 4 bones that belongs to the current vertex, with 4 belonging weights. These are static and are buffered to the OpenGL context when the `Mesh3d` is created. At each frame when an animated model is rendered, pre-calculated bone transformations (see 5.5.2) for the current frame is sent to the shader uniform `bones[100]`. A maximum of 100 bones can be uploaded per mesh, however, the actor shown in Figure 22 only uses 19. We have the uniform `hasAnimation`, to tell the shader if it should use the bones-array or not.

```
    mat4 boneTransform = mat4(1.0);

    if(hasAnimation)
    {
        boneTransform = bones[boneIDs[0]] *  weights[0];
        boneTransform += bones[boneIDs[1]] * weights[1];
        boneTransform += bones[boneIDs[2]] * weights[2];
        boneTransform += bones[boneIDs[3]] * weights[3];
    }

    vec4 skinVertex = boneTransform * vec4(vertex, 1.0);
    gl_Position = mvpMatrix * skinVertex;
    ...
    vec4 skinNormal = boneTransform * vec4(normal, 0.0);
    Normal0 = (modelMatrix * skinNormal);
    WorldPos0 = (modelMatrix * skinVertex);
    ...
```

Figure 17: Use of bone transformations in the shader.

For each vertex that is rendered, the shader will look at the four bone IDs to see which bones affect the vertex. The IDs are used as indices into the `bones[100]` array where the shader finds the bone's transformation matrix. The weights tells how much each bone affects the vertex. This will be used to offset the vertex and the belonging normal by multiplying them with the bone transformation matrix and the weight (see Figure 17).

Information about the animation to be played is saved in the Actor's graphics component, as mentioned in 4.3.2. On each frame, the `stepAnimation()` is called to step the time of the animation forward. The time is used to fetch the right set of bone transformation matrices from the component's mesh before passing them to the shader. See Figure 18.

```
void ActorGraphics3d::stepAnimation(const nox::Duration& deltaTime)
{
    this->currentAnimationTime +=
        (util::durationToSeconds<float>(deltaTime)
        * this->currentAnimationSpeed);
}
```

Figure 18: How animation time is incremented/decremented for each frame.

The code in Figure 19 shows how the bone transformation matrices for one model are passed to the shader for each frame.

```
for (unsigned int i = 0; i < boneTransforms.size(); i++)
{
    std::string name = "bones[" + std::to_string(i) + "]";
    GLuint boneLocation = glGetUniformLocation(renderData.
        getBoundShaderProgram(), name.c_str());
    glUniformMatrix4fv(boneLocation, 1, GL_TRUE, glm::value_ptr(
        boneTransforms[i]));
}
```

Figure 19: Passing bone transformations to the shader.

5.3.5 Deferred rendering

The `onRender()` function in the `OpenGlRenderer3d` class renders the whole world to the window. It traverses the root node and renders the objects in the scene graph together with the lights added by the light render nodes described in Section 5.3.1. The rendering is divided into three render passes; geometry, stencil and light. These render passes renders into the `GBuffer` which manages a framebuffer object with multiple texture buffers attached to it. Each texture buffer stores data about a temporary texture. The depth texture buffer is a texture containing depth data about what the camera sees and is used by OpenGL to determine whether an object is in front or behind another. The geometry pass traverses the root node and renders all the objects colour, normal and position data into their own respective texture buffers as well as populating the depth texture buffer inside the `GBuffer3d`. After the world is rendered in the geometry pass, it will use the created textures to render a lit world. This is done by looping through all the lights and render them using the stencil pass followed by a light pass. The stencil pass do not render to any texture buffer, but it creates a stencil buffer, like a paper with holes, used to delimit the light volume so the light will not affect objects outside of its range. After the stencil buffer is created the light pass renders the light and the objects that are affected by it into a final texture in the `GBuffer3d`. After the lights are rendered into the final texture it is blitted to the screen.

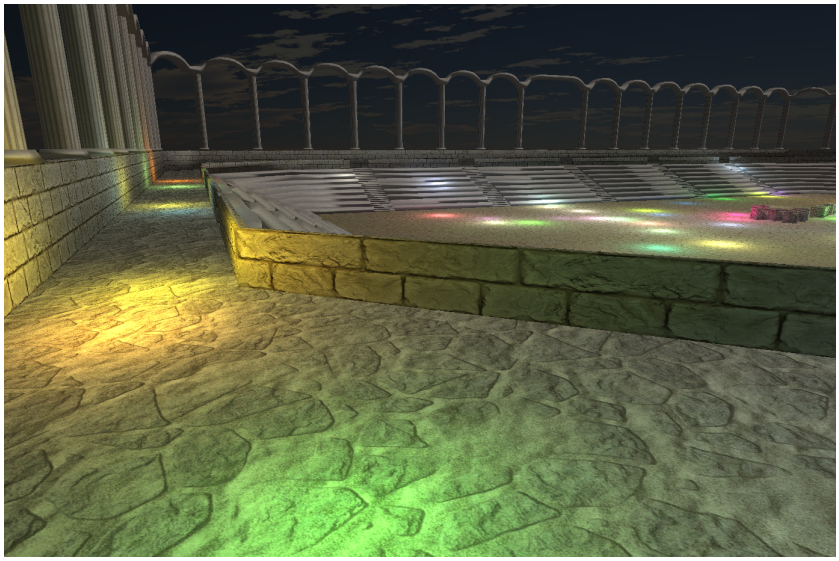


Figure 20: Deferred light rendering makes it possible to have many light sources without significant performance loss.

5.3.6 Transparency

Figure 21: The first picture shows a transparent grid. The Second picture shows two transparent object without proper sorting.

Rendering transparent objects requires additional work compared to rendering non transparent objects, this is due to the fact that the transparent objects needs to be rendered in the order back to front to get the transparency to look right. They also needs to be rendered last after every non transparent objects are rendered. To solve this problem the drawing of the transparent and non transparent objects are separated by storing all the transparent objects in a separate vector. The non transparent objects are rendered first

when traversing the scene graph, and the transparent objects are drawn at the end of the frame. If the vector was not sorted correctly on distance from the camera, the result would be as shown in Figure 21.

5.4 Actors

Everything that exists in the 3D world is called an actor. An actor can be anything from a player, a house or a simple light source. The actors are empty containers for components describing a functionality like health, graphics, light and animation. Every component needs to implement the following functions: `initialize`, `serialize`, `onUpdate`, `onCreate`, `onComponentEvent`. The actors are created from JSON files 4.3, see the Pyroeis document for detailed information [18].

5.4.1 Actor transform

The transform component handles the actors translation, rotation and scaling. The initialise function retrieves and stores initial values for these from actor's JSON file. The `getTransformationMatrix` function generates a transformation matrix using the position, rotation and scaling data, this function is used to get the model matrix when rendering the actor. When this component changes it broadcasts an transformation changed event notifying other systems.

5.4.2 Actor graphics

`ActorGraphics3d` is a component created for handling actors graphical properties like shape, animation and colour. This component contains four major data members; `meshData`, `actorTransformNode`, `renderTransformNode` and a `rendererNode`. The `meshData` is a pointer to a `Mesh3d` object containing 3D model data. The remaining three data members are scene graph nodes used to render the model data. The `actorTransformNode` contains the actors transformation data 5.4.1. The `renderTransformNode` is appended to the `actorTransformNode` and is used to change or offset the actors original transformation. The `rendererNode` is appended to the `renderTransformNode` so it can use the transformation data from the transformation nodes when rendering the model data. The model data pointer is set when the component is created in the `initialise` function which provides a JSON object containing the name of the model and a path to its data. After parsing the JSON object it runs the asset managers function for loading in a model using the name and path as arguments, see 5.5. After the initialisation is done it will run the `onCreate` function which creates the three scene nodes and appends them in the right order and broadcasts an event telling the renderer to add the `actorTransformNode` to the root node. The `onComponentEvent` function updates the `actorTransformNode` transformation data whenever the actor transform component is changed.

5.4.3 Actor light

Light sources are created and managed by the component `ActorLight3d`. When the component is created it will run the `initialise` function which provides a JSON object as a argument containing data about multiple light sources. It loops trough all the light entries in the JSON object, parses them, creates the lights and adds them to their own `LightRenderNode3d` object. After the `LightRenderNode3d` object is created it is added to a unordered map using the lights name as a entry index, it will also broadcast

an event telling the renderer to add the `LightRenderNode3d` object to the root node. After the `initialise` function is done it will run the `onCreate` function and get a pointer to the actors transform component which is used to update the light positions whenever the actor position changes in the `onComponentEvent` function.

5.4.4 Actor physics

The component that handles physics for an actor is called `ActorPhysics3d`. When initialised it tries to retrieve all information about its physical properties, type and shape from the JSON object. The information it cannot find is set to a default value, usually 0 or an empty string. With the exception of certain properties, for example a dynamic object where it can not find the mass; the mass will be initialised to 1. Without it the object might just as well be a static object. You can read more about this in Section 4.3.1. The component has many get and set functions for the physical properties, these are used to control Bullet when stepping the physics world 5.6.5. The `ActorPhysics3d` component is interacting with the physics engine through the `BulletSimulation3d` class.

5.4.5 Actor control

Nox provides a key mapper that allows the game developer to map actions like "move left" or "move right" to different keys using JSON, see Section 4.3.4. When a key with a registered action is pressed or toggled the corresponding action event is broadcasted. Nox provides an abstract class for handling the action events produced by the key mapper. Every control related component needs to derive this class and implement the `handleControl` function which provides a reference to an action event allowing the component to handle the action. Two components were created for handling the directional and rotational control of the actors using Nox vector control. See the Pyroeis document [18].

Directional control

The directional control component handles the movement of the actors in x, y and z axis. The `initialise` function retrieves data about movement speed and whether the control should be relative to either the camera or the actor rotation using the JSON object provided in the function argument. If the control is not relative to the camera or the actor it will be relative to the OpenGL coordinate system. When the `handleControl` function retrieves an action event with the id "move", it will get the movement values represented as a vector3. If the relative to camera or actor is enabled it needs to convert the movement vector3 to the right coordinate system before storing it. The `onUpdate` function updates the actors physical properties by setting the linear velocity to the stored movement data multiplied with the movement speed.

5.4.6 Rotational control

The rotational control component handles the rotation of the actors in x, y and z axis. The `initialise` function retrieves rotation speed from the provided JSON object. When the `handleControl` retrieves an action event with the id "rotation" it stores the rotation values represented as a vector of 3. The `onUpdate` function updates the actors physics by setting rotational speed to the stored rotation data multiplied with the rotation speed.

5.5 Model loader

Assimp, which was our selected model loader, provided us with a fast and simple interface for reading 3D model files 1.2.2. This made it easy to load the files, but using the loaded data correctly was the difficult part. Assimp was the most documented loader that was found, however parts of Assimp had almost no official documentation, for example how to use it for loading animation. A few good tutorials were found on how to use it and how animations could be loaded. The YouTube tutorials by "thecplusplusguy" [42], and the website ogldev.atspace.co.uk by Etay Meiri [43] helped us. Regarding the skin animations, none of the group members had any previous experience with this, and the process was more advanced than we imagined. We ended up using parts of the code/structure from ogldev's tutorials [43]. Due the poor documentation on animations with Assimp, we saw no other option given our short time. Ogldev's code is free to use and uses the same MIT licensing as Suttung. Even though much of the code was used, it took time understanding and refactoring it to our purpose.

5.5.1 Basic model loading

Loading models with Assimp was done using Assimp's `Importer` class. The importer's `ReadFile()` member function takes the filename as an argument, and different flags such as `aiProcess_Triangulate` and `aiProcess_GenSmoothNormals` can be used to tell the importer to perform different mesh calculations and mesh optimisations before it is finished. The function creates and returns an `aiScene` object where all data loaded from the model file can be fetched: vertex data, normals, texture coordinates, material data, animation data and more. Assimp uses its own types of vectors, matrices, floats, and so on. All the data has to be converted to standard OpenGL Mathematics (GLM) before it can be used with OpenGL. All loaded data for a model is stored in the `Mesh3d` class, which also contains the information used for rendering, like VAO and mesh entries information.

Model loading optimisation

All loaded mesh/3D model data is managed by the `GraphicsAssetManager3d` class that contains a map of `Mesh3d` objects indexed by name. To start with, each mesh was loaded once per actor that was using it. This would mean that if 100 actors were loaded, all using the same monkey head mesh, the model file would be processed and stored in memory 100 times. This is obviously inefficient, so a solution to how one mesh could be used for several actors was needed. The solution was to store all meshes in a map. When a new actor with a 3D model is about to get loaded, it will check what asset the actor is trying to load, and if the asset's name already exists in the map, the loading will be skipped. Often the game developer wants to load several actors that uses the same mesh data, and this will definitely save her both loading time and memory. There is some overhead when looking up a map entry, but as this only happens during start up, it will not affect the gameplay experience.

5.5.2 Loading animations

Figure 22: Animated goblin with visible bone structure

The animation data that is stored in the model files is only key frames that the bones moves between. For example, in the first key frame the actor's arms are hanging down, and in the next key frame the arms are raised. There is no way to fetch each animation frame directly from the file. In order to render a smooth animation, bone transformation has to be calculated between the two key frames using interpolation. To save performance all the transformations between the key frames are pre-calculate when the game starts. During rendering, the bones can be fetched directly instead of being calculated at each frame. Since an animation looks the same every time it is played this was done. See Section 7.1.2 for memory usage when it comes to animations.

Some model files contains more than one animation, for example "idle", "walk" and "jump". For each animation the model has, a loop runs, increments the current "time" and calculates all the bone transformations at that point. All animation frames for an actor are stored in a `map<aiAnimation*, vector<vector<mat4>>>` inside the `Mesh3d` class. The inner `vector<mat4>` holds all the bone transformations for one frame, for example the first frame in "jump". This vector is kept inside another vector that holds all the frames for the animation ("jump"). The map holds all the animations, mapped to the original `aiAnimation` from Assimp, where the animation data is calculated from. Figure 23 illustrates how the first 4 frames of one animation with 4 bones are stored in a `vector<vector<mat4>`.

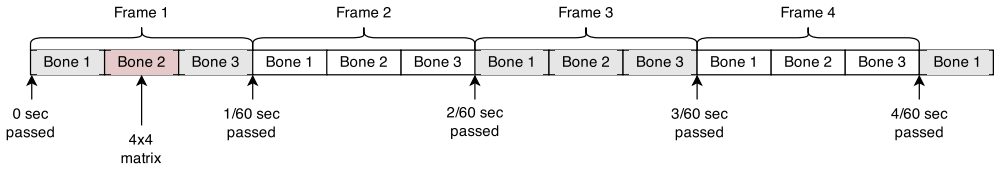


Figure 23: Storage of 4/60 seconds of an animation

The memory usage for each stored animation depends on the number of bones, vertices and length of the animation. For instance, the dance-animation of the goblin in Figure 22 lasts about 24 seconds. With a frame time of 1/60 second, 1440 frames will

be generated. With 19 bones and one 4x4 matrix per bone taking 64 bytes each, it ends up with about 1,75 megabytes for the 24 seconds, or around 73 kilobytes per second. Additionally there is also the bone data that holds which vertices that is affected by which bones. For that there are 4 integers for the bone IDs and 4 floats for the bone weights, per vertex. The goblin has 2880 vertices, so the total size of bone information is 2880*((4*4B)+(4*4B)) = 92 kilobytes. The graphics asset manager mentioned earlier will make sure that no 3D models are loaded twice - thus generating the animation frames will only happen once, even if the developer wants several clones of the actor. Figure 24 illustrates how animation frames are shared between actors.

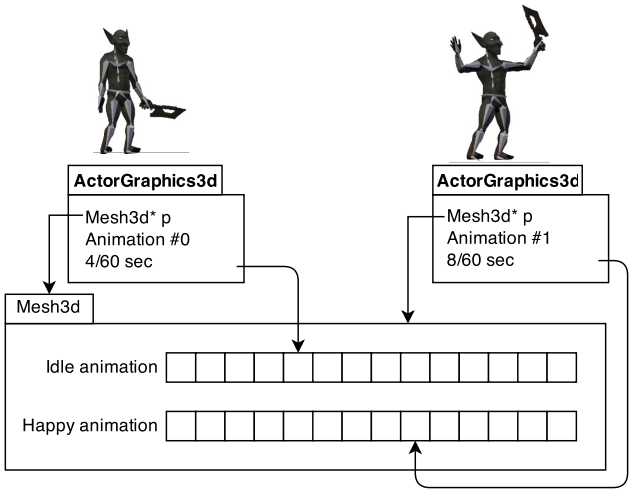


Figure 24: Two actors sharing the same Mesh3d, playing different animations

5.5.3 Textures

For loading and storing 3D textures, a third-party library was needed. For this the open source texture loader SDL_image was used. This was already included in the Nox engine, and all Noxplus members was familiar with it. It provides an easy API for loading and storing textures.

5.6 Physics module

An interface class, `Simulation3d`, was created for the integration of the physics engine. The `BulletSimulation3d` class extends this class and handles all communication with our chosen physics engine Bullet, with one exception. See Section 7.3 for why we chose this physics engine and Section 5.6.1 for the exception. The interface class contains a large number of member functions that all implementations need to implement. The reason for creating the interface class was to follow the requirements mentioned in Section 2.5.3, making it simpler to remove Bullet and implement another physics engine. The interface class that Suttung already had created could not be used since it is only meant for 2D physic engines.

5.6.1 Bullet physics library

Bullet has a class called `MotionState`, this is used for linking the renderer and the physics together. This class was derived to create a custom class that links the 3D ren-

derer with Bullet. This choice was made after some research and talking with supervisor Nowostawski, who said there are usually two ways to do it:

1. Pointer from the physics to the scene graph.
2. Pointer from the scene graph to physics.

The way that was implemented, with the `MotionState` class, is the same as point one in the list. A possible, if terrible solution could have been to updated the physics then loop through our scene graph and update the position and rotation of our actors with the information from the physics. This would have been a complex and inefficient way.

In `BulletSimulation3d` there is one instance of `btDiscreteDynamicsWorld` which acts as the simulated 3D world. It is able to simulate discrete physics, but in case soft bodies is to be implemented later it can be replaced with a `btSoftRigidDymaics-World` which is an extension of the previously mentioned class, as shown in Figure 25.

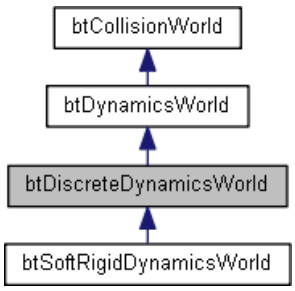


Figure 25: Bullet Collision world hierarchy. [44]

5.6.2 Rigid bodies

In order to simulate a world using Bullet, a set of rigid bodies with physical shapes must be created. This is done by first creating a `ActorMotionState3d` object which links the physics to the rendering. The function `setWorldTransform` runs when a body has moved. The `getWorldTransform` runs just once when the body is added to the world so Bullet knows the initial position. Lastly a collision shape is created, more about this in Section 5.6.3. Then an object is created that contains all the data a rigid body uses when being created. We pass in the `MotionState` object, the mass and the inertia of the actor, and its shape. With this object, the rigid body object is created and has its physical properties set. Now with an instance of the rigid body, a void pointer that Bullet has called `userPointer` is set to point to a struct we made that contains a pointer to the actor and a pointer to the rigid body. This is used for many things, but the most important is the collision handling. A reference of this struct is retrieved from a map using the ID of the actor and have the user-pointer point to it. The final step is then to add the rigid body to the physics world.

5.6.3 Collision shapes

A collision shape is needed to be created for actors in order to simulate the world properly. These shapes can be basic shapes, like a box, cylinder or sphere, or they can be shapes formed by the vertex positions in a mesh. What type of shape an actor should use is read from the actor's JSON file, together with other physical properties such as mass, position and friction. This is used to create a `btRigidBodyConstructionInfo` which is passed as an argument when the rigid body is created. The collision shape is used for

detecting collision and needs to be created outside Bullet and passed into the physics world with the rigid body. A collision shape can be reused for multiple objects and Bullet suggests to do this with the more complex shapes [45]. Convex hull shapes does just this, as long as the objects have the same name in the JSON file.

Convex hull shapes

As mentioned in 4.3.1, the mesh is needed to create this collision shape. Before creating a new shape, it checks a map that contains all the convex hull shapes by name, if it is found it will reuse the shape. If the shape has to be created, it needs to retrieve the mesh. It uses the actors ActorGraphics3d component for this. This is where the mesh is stored. When retrieved the indices and vertices are used to create a `btTriangleMesh`. This is then used when creating a `btConvexTriangleMeshShape` object, which is set to be a convex hull shape. If the optimise Boolean is passed as true, Bullet will optimise the shape by reducing the number of vertices before the new shape is set. If not, the new shape will be used directly. When the shape is created, it is added to the map so it can be reused later.

Concave hull shapes

The concave hull shape also use the mesh to create a collision shape, however this type of shape can only be given to a static object. The mesh is retrieved and the shape is created in the same way as it is done for the convex hull shape 5.6.3, it is just set to be a concave hull shape object instead.

Compound shape

When creating a compound shape it will use the `createShape` function for every shape the actor has. It is always using dynamic aabb tree to accelerate early rejection tests. It can be composed of all the before mentioned shapes 4.3.1 4.3.1, with the exception of the static ones. The concave shape is an exception to this and it will let you move it around, like it was dynamic.

5.6.4 Collision detection

Bullet is handling the collision detection and every time when Bullet has done its work it runs a function that the game developer set, (`myTickCallback`). This functions iterates over all the contacts manifold, a contact manifold contains all the contact points for a pair of objects, meaning a collision has happened. Whenever a collision pair is found that does not come from the last time Bullet ran, a function is run to add this pair. Then a comparison with the new collision pairs runs with the ones from last round and check for matches, if there are no match for some of the old, it means that the collision has ended. The function to remove collision pairs is then called. Lastly the new collision pairs are updated to be the old ones.

In the function for adding new collision pairs we first check if any of the objects that are in the collision are non-actors, if so nothing is done. At the time of writing nothing is done with non-actors. If they are proper actors the sum of the normal and friction forces are calculated and, depending on how the developer has set up the collision callbacks either; run through the collision callbacks that have been set by the developer for the actors or send out an event with the information and let the developer handle it. If the function to remove collision pairs are called the same things are done as if there was a new collision the difference is that the forces and contact points are not found and

passed.

Collision callbacks

There were two possibilities when it came to implementing the collision callbacks. One way was to set up an event system so that when a collision was registered an event was broadcasted to the system and the developer would set up a listener and choose what he wanted to do with the actors. The second possibility was to set up a map with the collisions callbacks that the game developer could add the callbacks into. It was decided that both choices should be implemented so they could be tested against each other, and so the developer could choose the preferred one. In the end we decided to keep both and differentiate between what one to use with a Boolean.

The collision callbacks are stored in two maps, an actor ID is required to be passed with the callback together with what type of callback it is. There are two kinds of callbacks, one for when the collision stops and one when the collision starts. A map is used for each type so that the two types of callbacks are not mixed. The callback is given an ID for the map inside the first map. A second map is used for every entry in the first map so a game developer can set multiple collision functions. The ID for the callback is returned when first added in the map. It is also possible to remove a collision callback with the ID for the callback, together with the ID for the actor and what type of collision callback it is. If the callback to be erased is found, the remove-function will return true, and if it fails it will return false.

5.6.5 Physics functions

There are several set and get functions for the physic, that can be are accessed for each actor through their actor physics component. See Figure 6.

5.6.6 Debug renderer

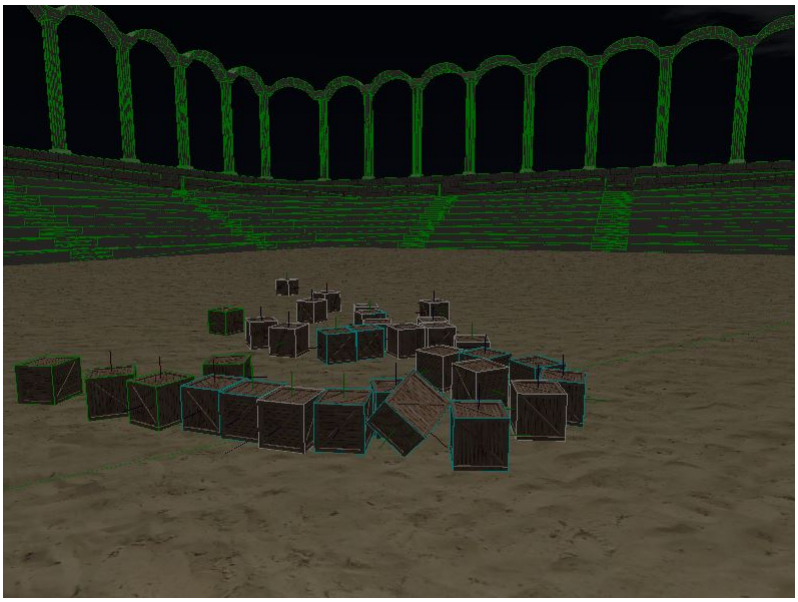


Figure 26: Rendering of the physics collision shapes.

Bullet provides a simple API for rendering the collision shapes geometry and represents their state using colours. A class that inherits from the Bullet's interface class `btIDebugDraw` was created and called `BulletDebugDraw3d`. The interface class provides an essential function `drawLine(from, to, colour)`. The `drawLine` function represents a line where the starting and ending point is the arguments from and to. For each collision shape this function is called multiple times creating the collision shapes geometry line by line. The lines received from the `drawLine` function is stored in a vector using a struct containing the starting point, ending point and the colour of the line. At the end of each frame the lines stored inside the vector is rendered using a simple shader which transforms the vertices and sets the colour. The initial way of doing the debug rendering was to render one by one line each time `drawLine` was called, resulting in many additional OpenGL draw calls.

5.7 Timeline manipulation module

To create the time manipulation module and its features 2.1, the gameplay data had to be stored somehow. This was discussed extensively and after some days of prototyping, we decided that the safest and most flexible way to do it was to store the data in a list. Read more about the prototypes in Section 5.7.5.

The focus was on storing data in a way that it could be accessed as fast as possible when it was needed. By storing and accessing the data continuously in the memory, the CPU cache would be utilised and we would get a slightly faster look-up of data.

5.7.1 Logging and storage

Logging and fetching data is handled by the world logger, `DefaultWorldLogger3d`. All frames are stored in a list of `WorldState` structs. Attaching and removing parts of a list is easier than with an array or a vector. Each world state contains multiple vectors storing all changes that happened since the last frame, see Figure 27.

```
struct WorldState
{
    std::vector<ActorPhysicsState> actorStates;
    std::vector<ActorAnimationState> animationChanges;
    std::vector<ActorAddedOrRemoved> actorsAddedOrRemoved;
    std::vector<TimeManipulationData3d> componentChanged;
};
```

Figure 27: WorldState containing all changes that happened in a frame

The structs `ActorPhysicsState`, `ActorAnimationState`, `ActorAddedOrRemoved` and `TimeManipulationData3d` are all used by the world logger. `ActorPhysicsState` stores the actor's physical properties such as position, rotation, scale, velocity and forces. It also holds a pointer to the actor's physics and transformation component so these can easily be accessed when the list is traversed. `ActorAnimationState` contains information about an animation change; the ID, time and speed that was active before the change, and the ID, time and speed that it was changed to. The old animation information is required when the time manipulation manager is rewinding, and the information about the next animation is used during replay. It also has a pointer to the actor's graphics component so the current animation data can be accessed and changed

quickly. An `ActorAddedOrRemoved` struct tells that an actor is added or removed at the current frame. It stores an `Identifier` that tells which actor is added/removed and a Boolean that tells if it was removed or added. `TimeManipulationData3d` is used for logging custom made components, more about this in the upcoming Section 5.7.4.

When the `DefaultWorldLogger3d` is created, it creates a buffer world state node. If changes are made during the first frame of the game, all these changes are saved to this node. At the end of each frame the member function `setEndOfCurrentFrame()` is called. Here the filled buffer world state is pushed back on the list before a new buffer world state is created, see Figure 28. The new node will be used to save all the changes for the next frame. The `frameIterator` is a list iterator that always points to the current world state and is used during rewind and replay.

```
void DefaultWorldLogger3d::setEndOfCurrentFrame()
{
    if (this->rewindEnabled == false && this->frameIterator == this->
        savedFrames.end())
    {
        savedFrames.push_back(this->bufferWorldState);

        // Create a new one to be filled:
        this->bufferWorldState = new WorldState();
        this->frameIterator = this->savedFrames.end();
    }
}
```

Figure 28: The setEndOfCurrentFrame function that runs at the end of each frame.

Logging of physical changes is done in the `DefaultWorldLogger3d` class, by the member functions `logActorState`. This is called from the `ActorMotionState3d` class' `setWorldTransform` member function which is automatically called by Bullet each time a collision body moves.

The world logger extends Nox's interface `IListener` and is set to listen for the events `SceneNodeEdited` and `AnimationChanged3d`. The `SceneNodeEdited` event occurs when an actor is added or removed from the scene graph. `AnimationChanged3d` happens when the actor's animation is changed, for example from "idle" to "run", in its graphics component. When the `AnimationChanged3d` or `SceneNodeEdited` events occur, a new entry will be stored in their respective vector in the current world state node.

5.7.2 Rewind and replay

Figure 29 illustrates how the data is stored in a list. During rewind, the engine will step backwards in the list, node by node, read the changes that is stored in the node and apply them to the physics component of the actors that the changes belongs to. When the replay is running, the physics will be applied just like during rewind, but it will also clear the data in the node so new, updated data can be saved. No nodes will ever be deleted, they are just updated. Because the changes are set in Bullet directly, these changes will immediately be sent to the logger and will replace the deleted data. Sometimes, new actors are spawned during replay, and in that case, these will be logged in the current node, together with potential interactions with the world. When rewinding

or replaying the animation it will use the stored animation data inside the current node. It applies the changes by using the pointer to the graphics component. As mentioned in Section 5.7.1 both the information about the previous animation and the new animation is logged every time an actor changes animation. During rewind it will set the graphics components animation id, speed and time to the stored "previous" data. If it is replaying it will use the "current" data. If the node has some data about removal or adding of an actor it will send out a `SceneNodeEdited` event to tell the engine to remove or add the actor from the scene graph. If custom created components has been logged on a frame, the component's onRestore will be called. See 5.7.4.

Figure 29: Shows the three first frames of gameplay, stored in a list

5.7.3 Paradox/conflict solver

The class `DefaultConflictSolver3d` is created to help the game developer solve the paradoxes or conflicts that happens when changes are done to the past, see Section 4.3.5.

One type of time conflict is direct interaction with the past. For detecting these conflicts, Bullet's own collision detection is used. A flag in the actor's physics component tells whether the actor is being simulated or replayed. When the actor is created, this flag is set to true, telling that it is simulated by the physics world. During rewind, all actors will get their simulated-flag set to false. New actors that are spawned has no logged data and will be simulated by the physics during replay. When a replaying actor is hit by a simulated actor, the actor that is being replayed will start being simulated too. When Bullet detects a collision between two actors during replay, it will first check if one of the actors have their flag set to true. If so, it will launch the conflict solver function if it was set, for the actor that is being replayed, before it sets it to start simulating.

In addition to the actor-specific conflict functions, there are also three other functions that the game developer can set. These are mentioned in Section 4.3.5 and works the same way as the actor-specific functions.

5.7.4 Logging of components

After some discussion with the group we came up with a few ideas for logging the components. One of the main problems was that we did not know how to handle logging of unknown data which the developer creates in each custom created component. Using template and interface classes was tried as a solution to this problem, but it did not work. How could the data be stored in an efficient way in the world logger or stored at all, when the data types that had to be stored was unknown? After consulting Nowostawski the problem was solved with a serialisation technique where the user needs to implement two functions where the data is being marshalled and unmarshalled. The data could then be stored continuously in memory as binary data or JSON. Since JSON adds

overhead, the decision to store it as binary was made. The `TimeLoggingComponent` which is an abstract class derived from the component class was created. Every component that wants to be logged need to derive this class instead of the original component class. The `TimeLoggingComponent` class provides three additional functions for solving the logging and restoration of the component; `logComponentData`, `onSave` and `onRestore`, see Section 4.3.5 for how they are used. The first function initiate the world logger's `logComponentData` function and is called whenever the component data has changed. The second and third function saves and restores the component using the class `TimeManipulationData3d` which is a container for storing and restoring component data. This container has two functions for saving and restoring data; save and load. The save function saves the data as raw data inside a char vector. When a component is logged the world logger creates an object of the container and stores it the current world state before passing it down to the component using the component's `onSave` function. The load function takes a pointer to the object that wants to be restored and copies the raw data stored inside the vector to the pointer's memory address. When the component needs to restore the saved data, the world logger will pass the same `TimeManipulationData3d` object to the component using its `onRestore` function. `onRestore` is called by the world logger during rewind and replay.

5.7.5 Discarded prototypes

First prototype: One array

The best way to store the physical changes was to store them as vectors instead of full model matrices, this was found during the implementation of the first prototype. This increased the time to store them away and use them because of the converting back and forth. However the choice was base memory usage concerns.

A `struct` containing the data for one actor-change was created to store all changes sequentially as single entries in the array. The `struct` would have a Boolean flag to mark where each frame ends; the last stored entry in the array would get the flag set to true when the frame finished. Having unused Boolean entries in each `struct` would not take up noticeable amounts of memory. With one frame per entry where all actor changes was to be stored, space had to be prepared for each frame. How much space was needed for each frame would vary, based on how many actors that was in the world. Either the developer would have to set it or there had to be a pointer in each `struct` telling where the data is stored outside the array. None of these was an option. Pointers would fragment the data and we would loose performance having the data spread all over in our memory. Having unused entries would not be great either.

Once the basic logging, rewind and replay was working the team started thinking about some things that had to be taken care of. The thing was that the physics was stopped updating during rewind. This would become a problem since the physics had to run for collision detection and updating objects that were to be updated during playback and rewind.

New actors spawned before playback would not be logged before the playback was done. When the playback was finished, the logging of the new actor started, but this was saved in the end of the array. What the team wanted was to log the newly spawned actor and append this data to the frame area where it belongs in the array. Then, during rewind, the array could be looped sequentially and the stored changes applied.

There were several ideas how to solve it:

1. Convert the array to have pointers. This changes the memory from continuous memory to scattered, this will increase time for retrieving data and this was not optimal.
2. Sort the array so new data could be inserted in a frame specific frame. This is time consuming and sorting an array for 2 million entries would take about half a second. It could be done in a separate thread and skip a frame for every time new data for a new actor was to be added so the child thread had the time it needed to do the insertion.
3. Use the current array and logging method and create an external list with nodes sorted on frame number, pointing to the right elements of the array. This would maintain the continuous storage of data, but the look-ups in the array during rewind/playback would not be fully sequential. Traversing it would cause possibly a lot of cache misses on the CPU, so performance wise it would basically be the same as storing all the data in a linked list.
4. Loop through the array once before new actors is spawned and reserve space for the new actors in each stored frame. The problem with this method is that it's newer known if the new spawned actor is going to move/change it state every frame resulting in potential unused entries in the array.
5. Let the user set the amount of new actors that are to be spawned in a game. So an amount of needed array elements can be reserved when the game starts. The same problem arises as the previous point. If the actors are to move around on every frame is unknown. If no actors are moving there would be a lot of wasted space.
6. Log the new data at the end of the stored data, while playing the data from the beginning of the array and set this to be overwritten by new data later. When all this space has been overwritten, the logger can continue writing at the end of the array again. In theory, it could work, but it would be a mess of indexes to keep track of, and the chance of bugs is large.

To start with, an array with 2.000.000 entries were created, just to make sure that memory was not a problem. Some calculations were done on how long it would take before the array was full. The following numbers were calculated out from one actor continuously moving and being logged 60 times per second:

33.333 seconds log time with 1 actor
16.666 seconds log time with 2 actors
11.111 seconds log time with 3 actors
8.333 seconds log time with 4 actors
6.666 seconds log time with 5 actors
166 seconds log time with 200 actors

The program would not allow any larger array than this. A possibility to circumvent this would be by storing the array globally, but keeping our data secure took priority.

Second prototype: Two arrays

The second try was none of the above but a completely different idea. There would be two arrays, one used for logging and one used for playback, at one time. The logging

would start like before on the first array, and when rewind started it would play backwards on that array and apply all the physical states that is stored here, to the world. When the rewind stops and the replay starts, it would swap array and start logging on the second array while reading/playing the data from the first array.

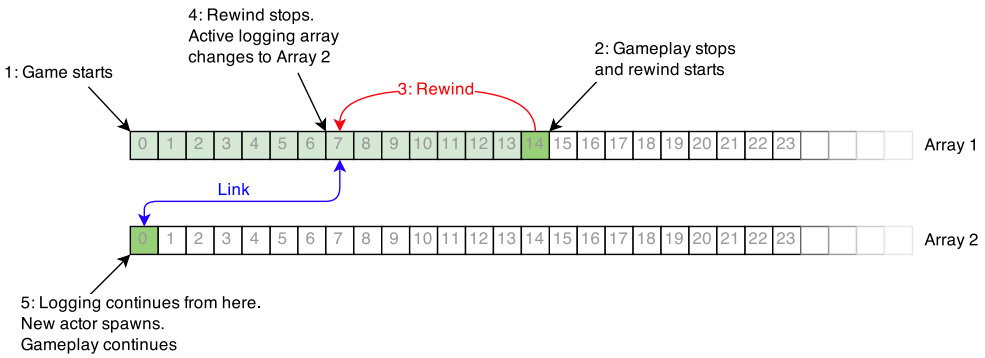


Figure 30: Logging data using two arrays: first rewind

In Figure 30, when the active logging-array change from array 1 to 2, a link is created between the two arrays telling what index the logging will start at in array 2. This item in array two, index 0, will be marked with the index of where array 1 left, which was index 7. This way it would know where to swap array when looping trough the arrays during rewind or playback. In this example, a new actor is spawned after the player stops rewinding and gameplay continues. This actor will be able to do changes in what was previously logged. The new actor will be logged on the second array, together with the old gameplay that was stored on the first array. This solution gives us continuous memory and good caching utilisation. The only problem is that there are now 2 arrays taking up the double amount of memory.

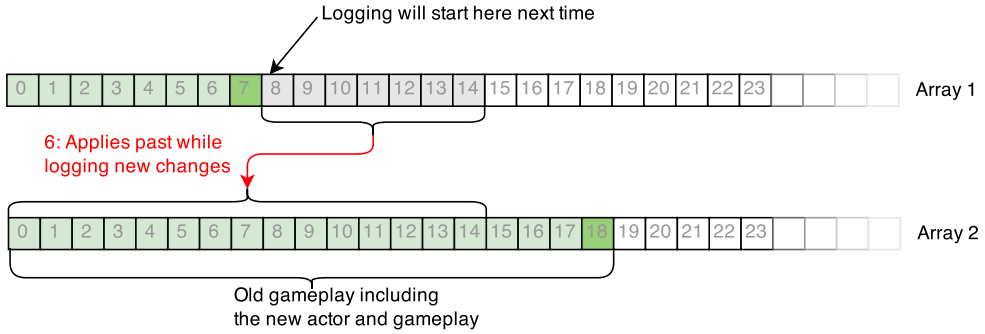


Figure 31: Logging data using two arrays: replay

This option works for straight forward rewind and playback, but when the player starts rewinding before all the old data has been saved, this data will later be overwritten and lost. The team tried to find a way to handle this by storing different indexes, but

realised that the complexity would make the Time manipulation module weak and it would not reach the requirement of robustness 2.3.

5.8 Control system module

5.8.1 Camera controls

The camera controls must be implemented by the developer in an update function, but the functions that are to be called to move the camera itself are in the `Camera3d` class. There are 4 functions that control the direction of the camera; `moveForward`, `moveBackward`, `moveLeft` and `moveRight`. The left and right functions moves the camera from side to side, instead of actually rotating the camera. Camera rotation is done with the mouse, it finds the length of how far the mouse has moved, if it is less than the max amount, a new view direction is calculated. It is also possible to reset the camera in case the developer becomes lost, through the `reset()` function.

5.8.2 Time control

- Rewind - Forward replay - Pause

5.9 Demo

6 Testing

6.1 Performance testing

The performance tests were run on a computer with the following specs:

- Operating System: Microsoft Windows 7 Ultimate 64 bit.
- CPU: Intel Core i7 920 @ 4 GHz.
- GPU: NVIDIA GeForce GTX 560 Ti.
- RAM: 16 GB @ 1600 MHz.

6.1.1 Rendering without physics

To test the rendering performance several stress tests were performed. In the first test a wooden textured box with a number of 356 polygons were used. This was cloned and placed to form a wall of boxes. All boxes were visible during the test and all physics were disabled. Figure 32 shows how the frame rate drops with different amount of boxes.

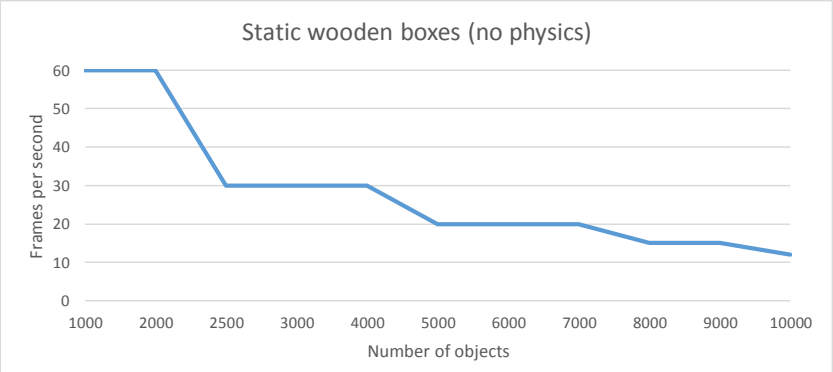


Figure 32: Performance when rendering static low-poly boxes without physics.

A test was also done with a more complex model. This model has 16 912 polygons which is a lot more than the wood box. The frame rate starts to drop at about 100 objects, which corresponds to 1 691 200 polygons vertexes. With the wood boxes, the frame rate starts dropping at 712 000, which means that drawing more polygons per object is most efficient. See Figure 33.

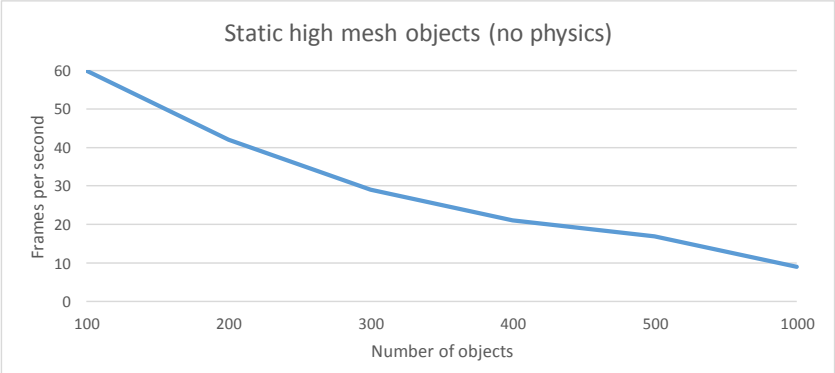


Figure 33: Performance when rendering static high-poly meshes without physics.

Animation performance was tested using multiple actors playing the same animation. From the tests we saw that the frame rate was varying a lot while the animations were playing, and apparently it depended on what part of the animation that was playing. The chart in Figure 34 shows the maximum and minimum measured frame rate during the tests. The frame rate did not start to drop before the number of goblins passed 20.

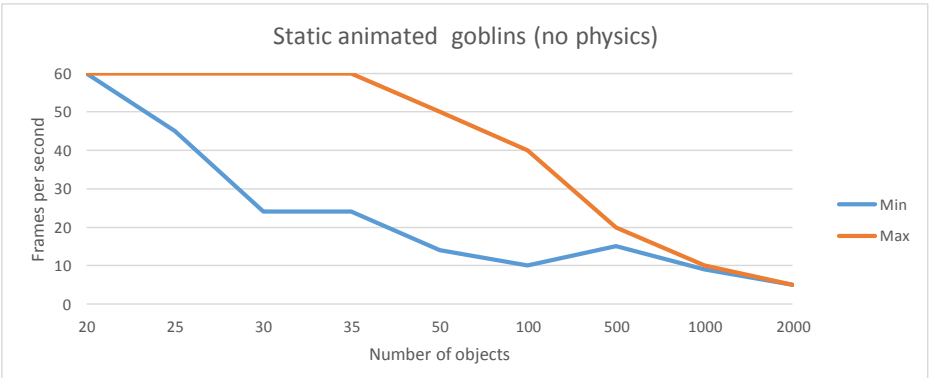


Figure 34: Performance during animation rendering.

6.1.2 Rendering and physics

The first physics test was done by placing boxes in a tower and let them fall. All boxes are using the simple "box"-shape. Each simulation ends when all boxes has landed and come to rest. The chart in figure 35 is based on the lowest registered frame rate during each simulation with different amount of boxes.

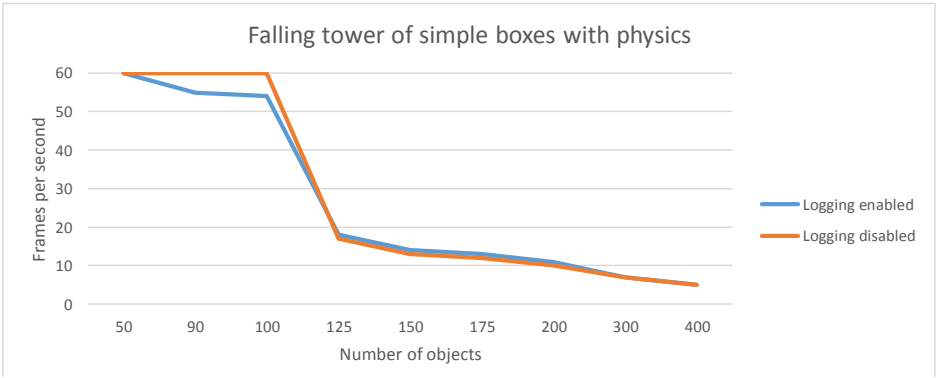


Figure 35: Performance chart of a tower of boxes falling.

The other physics test was done using convex hull shapes, also placed as a tower. A goblin figure with a number of 26574 vertices were used. The test was performed both with the optimisation flag set to true and false 4.3.1. Figure 36 shows how much performance is gained by using the optimisation flag, versus no optimisation. With two or more fully convex meshes colliding, the frame rate will drop significantly. This is however dependent on how many vertices the model has. The test was performed with and without a logging time manipulation manager, and according to the test, it was not noticeable performance loss when logging.

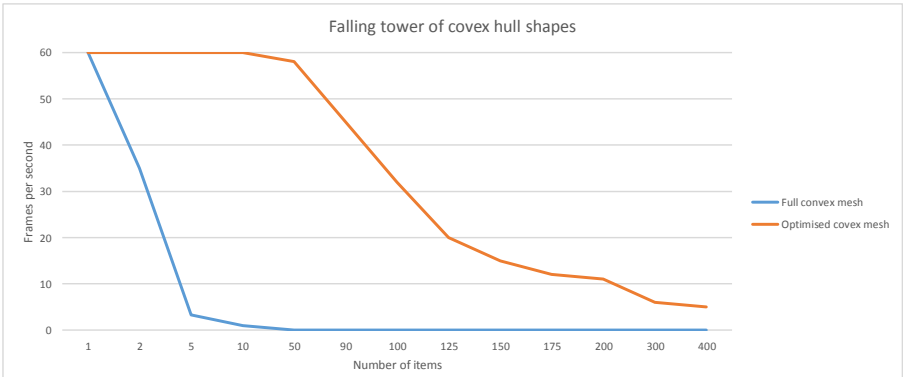


Figure 36: Performance chart of falling concave hull shapes.

Performance of simultaneous collisions between models was tested by placing a number of actors at the same spot before starting the physics. Also this test was run with and without logging. The lowest registered frame rates were used to map the performance. As shown in Figure 37, the logging does not seem to have any effect on the performance as long as the number of collisions stays below 25. Normally no objects should start at the same spot, so Figure 35 gives a more realistic picture.

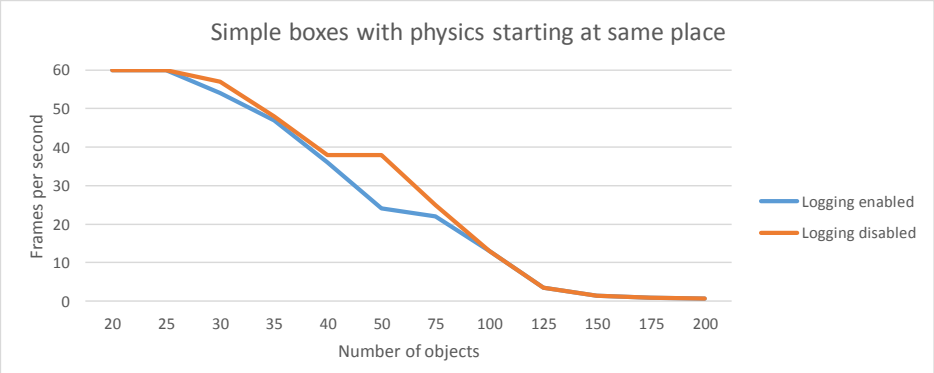


Figure 37: Boxes colliding simultaneously by starting at same spot.

6.1.3 Lights rendering

Noxplus uses deferred rendering to render lights. This is described in 5.3.5. To test how the light rendering performs during high load, multiple sources were placed in a square over a surface. See Figure 39. Looking at the test results in Figure 38, up to 400 light sources are rendered without problems. The radius of each light in this test was calculated to be 7.32. From 400 and up the frame rate drops next to linearly with the number of light sources.

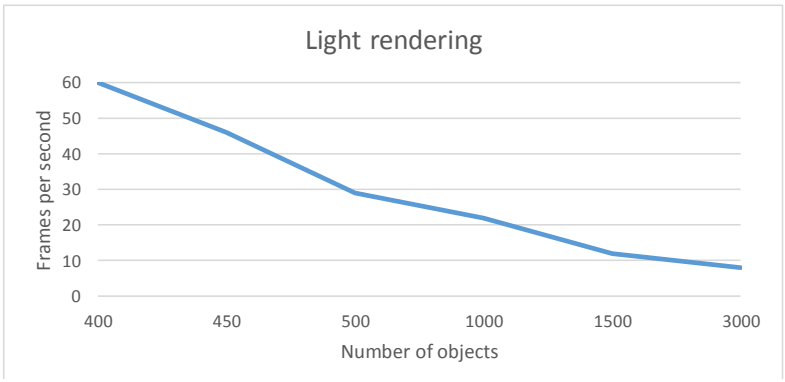


Figure 38: Performance chart of light rendering

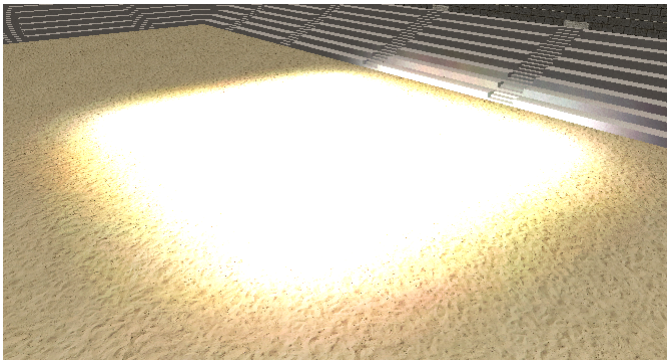


Figure 39: Stress testing deferred rendering

6.2 Unit testing

GoogleTest framework was used to do testing for specific cases. Tests was set up for the different modules; For the physics module three different tests were set up, where only one gave us some exceptions: When testing the creation of an actor body. It was found that we needed some checks in the ActorMotionState3d class to make sure that both of the required actor components were made. A mistake was found where we needed to make sure that the mass of a static object is set to 0. For the Timeline Manipulation module we found an index mistake. The logic tests all went fine without a single error or exceptions.

6.3 Linux and Mac

To follow the requirements of having Noxplus cross platform 2.5.4. Testing to make sure that the extension ran fine on both Linux and Mac had to be done. This was done with help from Suttung, as they had both Linux and Mac set up with the engine. They did so on our request at regular intervals. They then reported back on what they found at the next meeting. If they knew how to fix it, they did so and created a pull request that we then later pulled in to our repository.

7 Discussion

This chapter contains thoughts and ideas around what we have done and what we could have done differently if we were to do the assignment again. We also talk about things we tried for different problems and why we made the choices we did for implementation and design. In addition we talk about problems that occurred during the assignments. Possible solutions for these problems are also talked about.

7.1 Scene module

7.1.1 Rendering

When we first started looking into how we could implement the rendering system we found that there where two possible solutions. The first choice was to redesign the existing classes and functions, making support for a third Z axis. The second choice was to create our own separate rendering architecture in a similar way to how the current 2D rendering works. A decision was made to create a similar 3D rendering system, while we learnt how Nox worked. We would then refactor the Nox renderer to have support for a third axis. During our implementation and design of the 3D modules we found that the 2D rendering system in Nox was not as modular and flexible as we initially had thought. Refactoring the 2D rendering system to also support 3D would not be possible without rewriting the whole renderer. A new decision was made to use what parts we could from the 2D renderer and use what we had at the time. The decision was made on the basis of a meeting with Supervisor and Suttung H, to avoid ruining the functionality of the Nox engine.

The interesting outcome of this decision is that in theory it should be possible to use both the 2D and 3D rendering system together. Something like Sarepta studio did in the game Shadow Puppeteer [46]. This is however not tested and as mentioned is only a theory.

When creating the rendering system we had to chose between two different approaches; create a simple renderer and use it as a base for further development or design and implement the whole renderer at once. Because we wanted to start implementing the time manipulation module as fast as possible we chose the first approach and started with creating a really simple renderer we could use to render simple models and primitive objects such as cubes and spheres. With this simple rendering system we could start developing the time manipulation modules in the earlier stages of the development phase while making improvements and adding new features to the renderer.

7.1.2 Animation performance

One of the problems we encountered with the bone/skin-animated models was that they slowed down the rendering dramatically. In the beginning, on every frame, the bone structure of the model was traversed while it interpolated and calculated the actor's new "pose" for the next frame. Having one or two animated actors in the scene went fine, but when we added a few more the frame rate dropped a lot.

When we brought the problem to our supervisor Nowostawski, he suggested that we

calculated all animation frames before we started rendering the animation. This is how it's done in most other game engines too, so we decided to give it a try and found that this worked well. The pre-calculation of animation frames did not require any significant memory space 5.5.2. Considering the performance boost it gave us by not calculating all the bone transformations on the go, it was a good decision.

7.1.3 Multithreading

During the problem with animation and the frame rate drop we thought about threading the renderer. We believed the problem was that the renderer took too long. Threading the renderer is quite common since the CPU will spend most of its time waiting for the GPU to finish its work. Meaning that if we threaded the renderer the CPU will be free to do the physics update while the GPU does its own thing.

Another idea was to traverse the scene graph and make one thread per node that started calculating the animation for each actor asynchronous. When all threads were finished, the scene graph could be traversed and again and rendered. We were quite unsure about this idea, because so many threads would be created and deleted at each frame.

The problems we encountered however was not easily solved. The first problem was how would us threading the rendering affect the Nox engine? Would it ruin it for Suttung so they could not take in our extension? We brought it up in a meeting with them, where they said they had thought about threading the renderer, but they haven't had time to do it. And they did believe it would destroy things for them H. We also found a problem with the OpenGL context. If we were to thread the renderer we had to initialise OpenGL on the render-thread, since the OpenGL context needs to be on the same thread as the OpenGL specific functions are called. This would require us to move the initialisation of it and that would definitively destroy functionality for Suttung. We decided against threading the renderer and let Suttung handle it, we would rather try and find another solution. The thought turned away from threading and towards optimising the current extension. This would go against the principle from Extreme programming; to do optimisation in the end of the construction phase, but we did not see any other solution 3.1.3.

7.1.4 Transparency

We did have transparency implemented in the beginning of Sprint 4, see Section 5.3.6, but when we later implemented deferred rendering, we had to rewrite the rendering process. The transparency rendering where down prioritised and we did not get the time to re-implement it. This should not have been implemented before later in the process when we were sure that what we had would not change. The sources we found also says that transparency is very difficult to implement when using deferred rendering [47].

7.1.5 Deferred rendering

When implementing the light it was essential that the design would scale well with many lights without being the engine bottleneck. To achieve this we had to do some research and find a rendering design that would suit our needs. After doing some research we found that we had to decide between the two rendering designs; forward or deferred rendering. Forward rendering is the standard rendering technique that most engines uses, but the problem with this rendering technique is that it does not scale that well in the aspect of lighting compared to deferred rendering. This is due to the fact that

the numbers of lights calculations preformed by the forward rendering designs is heavily dependent on the size of the scene, the number of total light calculation needed is calculated by multiplying the number of geometry fragment and the number of lights in the scene. Because of this dependency it is really expensive to have lights in bigger scenes. The deferred rendering design decouples the scene geometry and the lights by rendering to a buffer before applying the light calculations using the the same buffer. The total numbers of light calculations needed is the size of the buffer multiplied with the numbers of lights in the scene, the buffer size is usually the same as the window resolution. We decided to use the deferred rendering design because it is more scalable and preforms far less light calculations on bigger scene than the forward rendering design [48]. Deferred shading is a widely explored area of 3D graphics, and we found a good tutorial on this that we based the implementation on using the same code/structure with some modifications [49]. The code from the tutorial is free to use and uses the same MIT licensing as Nox.

7.1.6 Camera

The camera has a known problem, being the famous gimbal lock problem, this should be solved with using quaternion instead of matrix rotation. It has been made as generic as possible so it is usable for many different scenarios.

7.2 Model loader

As the fundamental rendering began to finish, we wanted to add a little extra so we could give our demo a better look. We started looking at different asset loaders for loading 3D models into the engine. One option was "lib3ds". However, this only supported 3D-Studio's ".3DS" files, which is not convenient as a 3D Studio license is quite expensive. We also looked at the API of Autodesk, which is another well known file format, but we were more interested in a loader that supports multiple file formats. With that in mind, we came over Trimesh2 [50]. This is a multiplatform library with utilities for input, output, and basic manipulation of 3D triangle meshes. Trimesh2 were not well documented, it had not been updated for a couple of years and there is no git or CMake support. The final choice was therefore Assimp. To us this seemed like the most used loader, it supports over 40 different file formats, and it got more documentation. We also found a couple of good tutorials on how to use it.

7.2.1 Loading models

One problem with our implementation is that all model files are loaded when the first actor that is using the model is created. This means that when a mesh spawns in-game for the first time, the game will freeze while the model is loading. The only solution to prevent this is to spawn the actor somewhere the player cannot see it, and when it is supposed to be spawned, it can be moved to the right spot, or cloned. A better solution to this problem would be to make sure that all models are pre-loaded before the game starts. The engine could go trough the folder structure, look for files with specific endings and load them. This should be fairly simple to do if we add the loaded models to the existing map (Section 5.5.1) and change the engine to map on file names instead of the "name" specified together with the file name in the JSON file.

Another improvement when it comes to loading models would be to run each actor creation on its own thread. Then all actors could load asynchronously, instead of being

queued like they are now. None of the actors are dependent on each other, so synchronisation should not become a problem.

7.3 Physics module

We had to chose how to design the physics and look at the available physics engines. We knew we needed something that would work well with our time manipulation module. An engine that could re-simulate the physics exactly as it was every time after we rewind and started playing it forward normal again when there was no changes to the world. We looked at Havok, however it is not open source, a license is needed for it if a developer is to sell his game for more than $10 USD. This makes it unsuitable for our project because of our Licensing Requirements 2.7. Another engine we looked at was the Open Dynamics Engine (ODE). This is both open source, deterministic and platform independent [51]. Bullet is the last physics engine we looked at, it is open source and platform independent [17], and according to Nowostawski; deterministic. The reason we wanted the physics engine to be deterministic was because it was a requirement 2.5.3 for our time manipulation part so we could try re-simulation 4.3.5. We could not find any relevant differences between the two so the reason we chose Bullet was that it had support for OpenCL. We wanted to try running our physics on the GPU if we had the time. We could not find any information on if ODE has support for OpenCL or anything like it. Bullet is a well known, professional physics engine used in many video games [52]. It is also used for making special effects in movies, and NASA has used it for testing a new space robot project [17]. Bullet simulates our world's physical laws and provides us with advanced collision detection, and supports soft and rigid body dynamics.

Collision callbacks

What we wanted to implement was the possibility to run a single callback function, so that not every callback functions that have been set for an actor runs every time a collision occurred. When setting callback functions the ID of the callback is returned and the developer could then use this together with the ID of the actor when a collision occurred to specify what callback function is to run. At the point of writing this is only possible with removing the callbacks that the developer do not want to run.

7.4 Time manipulation

7.4.1 Working version: List and vectors

The last thing we implemented and went with can be read about in Section 5.7. Here we used a list where we created and added a node for every single frame. Due to the fact that no sorting or re-organisation are required in this implementation, we could have used a vector or an array instead. The reason we did not do this was to make swapping of data to the hard drive easier when this is implemented in the future. Attaching and removing parts of a list is easier than with an array or a vector.

All changes that happen in a frame will be stored continuously, however the storage of each frame will still be fragmented. This is because the changes for each frame is stored in separated vectors, see 5.7.1. In a small world with only a couple of actors, this is not efficient. On the other hand, in a small world, there is often few resources needed for physics and rendering, which will compensate. In a big world with a lot of actors, there are a lot of changes that happens at each frame and there will be more data stored in

each node, thus more data will be stored continuously.

We can not skip to different points in time in the prototypes. The rewind and replay happens sequential for all of them. This could be solved by creating snapshots of the whole world at regular intervals. We could then jump between them.

During research of time manipulation, we found a way to handle particle effects. Jonathan Blow the developer of the game Braid was the one who inspired this [22]. Implementing a way to store the seed that was used for the pseudorandom number generator, so that you would always get the same random values every time you ran it.

7.4.2 Swapping data to hard drive

One of the time manipulation features we thought of during implementation was rewind and replay time equal to how long the game had been running. To achieve this we could not depend on the memory alone, we had to figure out an efficient way to utilise the hard drive for storing of data. We discussed a few possible solutions and concluded with a design that we could implement, but we dropped it, due to the fact that it was not in our original plans. The idea was to always have three logical blocks of stored frames in memory at the same time; one "past"-block, one "current"-block and one "future"-block, each having a fixed number of frames. Figure 40 illustrates the same timeline going trough 3 different stages during replay. The little arrow is where the game currently is. When the current gameplay reaches the "future"-block, the "past"-block is written to disk and deleted from memory. The "future" block becomes the new "current" block, and data is read from the disk to become the new "future"-block. The blocks limits are decided by indexes pointing to the starting and ending nodes.

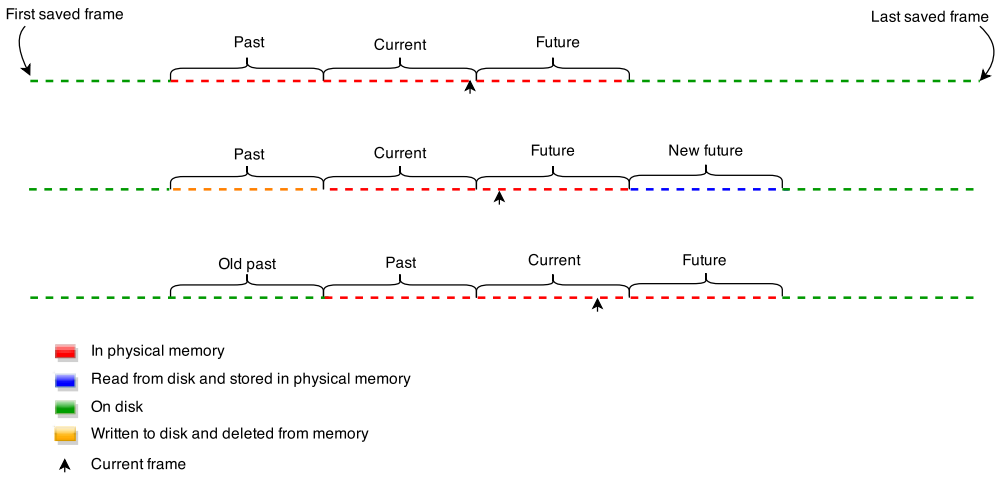


Figure 40: How data is swapped to disk during replay

7.5 Demo

Our initial plan was to create one demo for each of our modules, one for each of the milestones, and then merge them into one big demo. We always postponed this due to wanting to change our class names and directory structure 7.6, later it was postponed due to time. However, we did create a demo at the end of the project that shows all of

the functionality of our modules.

When creating the demo we had to decide how we wanted to do it. Where we making a small simple game or just some simple interactions. We decided to make things interesting and go for a simple game. Read more about the design of the game in 4.3.6 and implementation of the game in 5.9.

7.6 Development

7.6.1 Choosing assignment

At the start of the Bachelor assignment we had to make a choice between two assignments:

1. Customising a game engine to improve the performance in a specific development area.
2. Extending Suttung Digital's game engine, Nox engine, with the ability to represent the game world in three dimensions (3D).

The one we were sure we wanted to do was number 2. After some discussions with McCallum, Nowostawski and Suttung we came to a decision. We were going to try both and set a cutoff date where we decided if we where to continue onward with both or drop 1. A later decision we came to was; to never cut the time manipulation part since this would be where the possibility of the best grade was.

When we first were going to choose what engine we wanted to develop on both Unreal engine 4 and Unity 5 were not free to use before early March [53] [54]. They are also both large engines and would take time to learn. We were also focusing on technology and not developing games. One of the assignments also specifies that we are to extend the Nox engine with 3D capabilities. Our employer Suttung, was more interested in us working with Nox and having the developers of the game engine close by would help us a great deal. Choosing Nox over Torque3D had advantages and disadvantages. The first disadvantage was that Nox does not have the amount of support and documentation that Torque3D does. However the advantage was that we had the developer close by and could ask them questions on a regular basis. In addition, Torque3D already have 3D functionality implemented and would not give us anything to work with [55].

Since our assignment was quite large we had to narrow down our assignment. We did this by creating a list of what modules that where required and what we wanted to do as shown in Figure 1. The choices was made based on our research [40] and knowledge from previous courses.

7.6.2 Directory and file structure

During the implementation phase we discovered that we needed a new directory structure. Suttung's directory structure was not enough for the additional 3D extension. We brought it up on a meeting with them and they told us to make a draft. We sent them a draft on the 21. February, they did some modifications on it and said they would implement it. This took longer than anticipated by us, not before 25. of March was there a change from Suttung, and they had only changed the graphic folder and its subfolders. We then decided that we would make due and work with what we got.

During this period we also discussed about changing the prefix of our classes to a suffix. When we started we named our classes with the prefix "Trd", to avoid conflicts

with Suttung's classes. Both Suttung and supervisor thought a suffix "3d" would be better. This was changed after the "new" directory structure was received. This was also a big reason why we kept postponing creating our demo's since we would have had to rewrite them to work with the new structure and class names.

7.6.3 Re-prioritising

During one of our meetings in the later part of our project with Supervisor we decided to focus on time manipulation rather than focus on creating a nice API for developers. We agreed that the time manipulation module is a more important, exciting and new area within games so we decided to put our focus there rather than the API's.

A month before the end of our project, our supervisor Nowostawski suggested we re-prioritise our tasks relative to time manipulation. Focus should be put there instead of for example deformable mesh, since it is not an important feature, hard to implement, and should therefore have a low priority. We where also told to reorganise our plans and make sure we have the most important topics sorted for the next month.

7.6.4 Requirements

One of our requirements that we followed through the whole project was to have a physics engine that was deterministic. This was so that when we started on the time manipulation we could try using re-simulation instead of logging the whole world at every frame. We decided to not do this after some discussion with Bernt Tore Jensen and our supervisor. Read more about it in Section 4.3.5. However we did not remove it from our requirements in case an engine developer wants to try it.

Another of our requirements that we did not have time for was to make sure there are no significant drop in performance during heavy load. Our plan was to implement a "trivial" flag for the actors. They would either be skipped during a physics update or skipped for collision detection. Another thing could be to also use the flag during rendering, so it will be skipped when rendering the scene.

7.6.5 Software Development Methodology

rewrite after Gantt this section may overlap with Development process, go through it and check The use of RUP in the project have had both good and bad sides. The most helpful artifact was the design class diagram. It showed us how the classes and the layers where working together to form the system as a whole. It was created after we found out that we had problems understanding how Nox worked. After it was created we realised that we should not have jumped into Nox without creating the artifacts that we had planed for. The revised risk analysis was also helpful since some of the risks we came up with actually happened, although they just minor ones. We where missing people for up to a week more than once, the persons had to work extra hours too make up for the lost ones. We also had problems understanding Nox as mentioned above we solved this by both creating some artifacts from RUP and we talked with Suttung. Suttung helped us by creating some demos to show the functionality of the Nox engine. The programming guidelines was helpful to keep everything we made consistent. The requirement specification and supplementary requirements also helped us with making choices at many points during the whole project. The negative side was that RUP is quite the extensive methodology and figuring/remembering what the different documents were for and how they were set up took too long. We believe that RUP helped as much as it hindered our

project, we could have focused on Scrum and have had the same results. We would still have created the design class diagram and the Use cases and their explanations. It could have been because we the documents a bit late and they were not solid. We fixed this during the project and they became more helpful.

Scrum was our most valuable development methodology. The backlog was updated every time we found a new task that needed doing. Sometimes we threw the task into the current sprint if we found it important and meant that it had to be done before another task in the sprint was started. The sprints was created almost every week and lasted for seven days, sometimes we found it better to extend it if we felt that there where enough work to last another week. The last month we extended the sprint to last until the end of the project, since we had work to do on the thesis. This helped us keep an overview over what kind of assignments we had before the end of the project. The retrospective meetings were a waste of time. We found our mistakes early but they where always down prioritised since we believe they were not that important. Closing and starting tasks was one of those things, our burndown chart looks terrible, but we never had problems with knowing what tasks needed to be done and what tasks where being worked on. The milestone reviews helped us keep our backlog up to date and gave us an overview of what tasks we had to do. Some months we had planned too many tasks but we managed to get through it with a good result. The sprint review was a nice asset, it helped us keep us on track when it came to what tasks that had to be done to meet our milestone and goal. Since all group members were available on Skype at all times, we found that the daily scrum were not needed. Everyone always knew what the others were doing at all times.

We are quite happy that we included Extreme Programming, all of the principles worked well for us. Especially the pair programming principle. We didn't always program in pair but when we had difficulties or started new larger tasks we set two on it until the most difficult part was done.

8 Conclusion

The Noxplus extension encompasses two main modules. The first was to create a time manipulation module for games in 3D. This was implemented with both rewind and replay functionality. A conflict solver was also implemented as a solution to possible paradoxes and conflicts between actors. Three functions was included for game developers so they can decide on what they want done when replay starts, when replay ends and when replay is interrupted. They can be left empty. New actors spawned in any of these functions will be logged. If game developers create new components they can also be logged if done correctly, read how in Section 4.3.5. We have not found a module like this in any other game engines and it creates many new opportunities for new game mechanics in games.

The second module was to extend the Nox engine with the possibility for 3D. This was done by implementing an asset loader, that can load multiple file types that are supported by 3D modelling programs for example: Blender and 3D Studio Max. However, only two of the file types Noxplus support, supports animation. The second functionality that was added was physics for 3D. The physics that was implemented in the Nox engine only supported 2D so we had to implement a physics subsystem that worked for 3D. We also implemented deferred rendering for rendering lights, models and debug geometry. We created a scene graph to organise the rendering of the scene. A control system was also implemented to control the actors and camera in the scene.

A demo was created to show the functionality namely: rendering, physics, asset loading, rewind, replay and conflict solving. It is used for demonstrating our work and as an introduction to Noxplus. We created the sky dome ourselves and borrowed the rest of the assets.

We are quite happy with the finished product, we managed to increase the functionality for the Nox engine and make it more attractive for both game developers and game engine developers as we defined in our goals 1.1.2. However there were some functionality that would improve Noxplus. These are listed in the Future work Section 8.1. We have become better programmers while working with this project, increased our knowledge and skill in both game engine programming and handling larger projects. We have never worked with c++11 features before, for example std::functions, lambda and move semantics. There where many problems that were encountered but we solved them to our best ability, like our solution to the time manipulation 5.7.

8.1 Future work

Due to the task being quite large we have found features during the project that would make the engine better. These are mentioned here:

- Rewind and playing forward is possible at different speeds, done by scaling the delta time calculated in the execution loop.
- Review what the Information Security group have done and apply suggestions we find relevant to Noxplus.

- Time manipulation for 2D.
- Combination of 2D and 3D rendering, using Nox's 2D rendering on top of noxplus's 3D rendering or reversed.
- Optimise for more actors in a scene by fully exploiting Bullet using its OpenCL support.
- Support logging of custom components that the developer adds to actors, for example health.
- Time manipulation for particle effects.
- Ability to turn off actors from logging, by using a flag.
- Support for triggers and complex collision shapes.
- Solving camera rotation using quaternion instead of matrices.
- Soft bodies for physics simulation.
- Soft shadow mapping.
- Reuse model textures by using a map.

Bibliography

[1] Rational unified process, best practices for software development teams (online). URL: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf (Visited 14.5.2015).

[2] Sommerville, I. 2011. *Software Engineering*. International Computer Science Series. Pearson. URL: https://books.google.no/books?id=l0egcQAACAAJ.

[3] Cmake (online). URL: http://www.cmake.org/ (Visited 15.5.2015).

[4] Introducing json (online). URL: http://www.json.org/ (Visited 15.5.2015).

[5] Digital, S. 2015. Nox engine google group. https://groups.google.com/forum/#!forum/nox-engine/. Visited May. 2015.

[6] Rational unified process (online). URL: http://sce.uhcl.edu/helm/rationalunifiedprocess/ (Visited 14.5.2015).

[7] The rules of extreme programming (online). URL: http://www.extremeprogramming.org/rules.html (Visited 14.5.2015).

[8] Bitbucket (online). URL: https://bitbucket.org/ (Visited 15.5.2015).

[9] Sourcetree (online). URL: http://www.sourcetreeapp.com/ (Visited 15.5.2015).

[10] Toggl (online). URL: http://www.toggl.com/ (Visited 15.5.2015).

[11] Jira (online). URL: https://www.atlassian.com/software/jira (Visited 15.5.2015).

[12] Jira (online). URL: https://www.sharelatex.com/ (Visited 15.5.2015).

[13] Google docs (online). URL: https://docs.google.com/ (Visited 15.5.2015).

[14] Microsoft visual studio 2013 (online). URL: https://msdn.microsoft.com/en-US/vstudio (Visited 15.5.2015).

[15] Teamviewer (online). URL: https://www.teamviewer.com/ (Visited 15.5.2015).

[16] Assimp, open asset import library (online). URL: http://assimp.sourceforge.net/main_doc.html (Visited 15.5.2015).

[17] Real-time physics simulation, home of the open source bullet physics library and physics discussion forums (online). URL: http://bulletphysics.org/wordpress/ (Visited 15.5.2015).

[18] Vik, M. B. & Sporaland, A. 2014. Pyroeis. http://hdl.handle.net/11250/216763. Visited, Dec, 2014.

[19] Hockenberry, J. 2013. Multiverse: One or many? http://www.worldsciencefestival.com/programs/multiverse/. Visited May. 2015.

[20] Multiverse (online). URL: http://en.wikipedia.org/wiki/Multiverse (Visited 15.5.2015).

[21] Burlew, R. 2003. Giant in the playground. http://www.giantitp.com/. Visited May. 2015.

[22] Braid (game), 2008, Microsoft Game Studios and Number None, Inc. Number None, Inc. and Hothead Games (dev.). URL: http://braid-game.com/ (Visited 15.5.2015).

[23] Dark chronicle (game), 2014, Sony Computer Entertainment. Level-5 (dev.). URL: http://en.wikipedia.org/wiki/Dark_Chronicle (Visited 14.05.2015).

[24] Paradox (online). URL: http://www.umich.edu/~engtt415/paradox/ (Visited 15.5.2015).

[25] Time travel (online). URL: http://en.wikipedia.org/wiki/Time_travel (Visited 15.5.2015).

[26] Pillars of eternity (game), 2015, Paradox Interactive. Obsidian Entertainment (dev.). URL: http://eternity.obsidian.net/ (Visited 15.5.2015).

[27] Coraci, F. Click. http://www.imdb.com/title/tt0389860/. Last visited May 2015.

[28] Gödel, K. Jul 1949. An example of a new type of cosmological solutions of einstein's field equations of gravitation. *Rev. Mod. Phys.*, 21, 447–450. URL: http://link.aps.org/doi/10.1103/RevModPhys.21.447, doi:10.1103/RevModPhys.21.447.

[29] Closed timelike curve (online). URL: http://en.wikipedia.org/wiki/Closed_timelike_curve (Visited 15.5.2015).

[30] Company of myself (game), 2009. 2DArray (dev.). URL: http://www.kongregate.com/games/2DArray/the-company-of-myself (Visited 15.5.2015).

[31] Blow, J. 2010. The implementation of rewind in braid. http://gdcvault.com/play/1012210/The-Implementation-of-Rewind-in. Visited May. 2015.

[32] Skow, B. 2013. Notes on the grandfather paradox. Last visited May. 2015, last modified 2013.

[33] Noxplus. 2015. Noxplus demo. http://hovedprosjekter.hig.no/v2015/imt/spill/noxplus/?page_id=43.

[34] Simple directmedia layer (online). URL: http://www.libsdl.org/ (Visited 15.5.2015).

[35] Opengl (online). URL: https://www.opengl.org/ (Visited 15.5.2015).

[36] Sdl_image 2.0 (online). URL: https://www.libsdl.org/projects/SDL_image/ (Visited 15.5.2015).

[37] M. McShaffry, D. G. 2012. *Game Coding Complete*. Cengage Learning PTR.

[38] F. Buschmann, R. Meunier, H. R. P. & M.Stal. 1996. *Pattern - oriented Software Architecture*. John Wiley & Sons Ltd.

[39] Briand, L., Morasca, S., & Basili, V. Jan 1996. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1), 68–86. doi:10.1109/32.481535.

[40] Gregory, J. 2014. *Game Engine Architecture, Second Edition*. A K Peters/CRC Press.

[41] Blender (online). URL: http://www.blender.org/ (Visited 15.5.2015).

[42] thecplusplusguy. 2013. Opengl glsl tutorial 6 - assimp 3d model loader (part 1: static models). https://www.youtube.com/watch?v=ClqnhYAYtcY. Visited Apr. 2015.

[43] Meiri, E. 2012. Skeletal animation with assimp. http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html. Visited Mar. 2015.

[44] Bullet collision detection & physics library (online). URL: http://bulletphysics.org/Bullet/BulletFull/classbtDiscreteDynamicsWorld.html (Visited 15.5.2015).

[45] Bullet (online). URL: http://bulletphysics.org/mediawiki-1.5.8/index.php/Main_Page (Visited 15.5.2015).

[46] Shadow puppeteer (game), 2014. Sarepta Studio (dev.). URL: http://shadowpuppeteer.com/ (Visited 15.5.2015).

[47] Rendering transparency in a deferred pipeline (online). URL: http://techblog.floorplanner.com/rendering-transparency-in-a-deferred-pipeline/ (Visited 14.5.2015).

[48] Forward vs deferred rendering (online). URL:
http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342
(Visited 14.5.2015).

[49] Meiri, E. 2012. Deferred shading part 1-3.
http://ogldev.atspace.co.uk/www/tutorial35/tutorial35.html
http://ogldev.atspace.co.uk/www/tutorial36/tutorial36.html
http://ogldev.atspace.co.uk/www/tutorial37/tutorial37.html.
Visited Mar. 2015.

[50] Trimesh2 asset loader (online). URL:
http://gfx.cs.princeton.edu/proj/trimesh2/ (Visited 15.5.2015).

[51] Open dynamics engine (online). URL: http://www.ode.org/ (Visited 15.5.2015).

[52] Bullet (software) (online). URL:
http://en.wikipedia.org/wiki/Bullet_%28software%29 (Visited 15.5.2015).

[53] If you love something, set it free (online). URL:
https://www.unrealengine.com/blog/ue4-is-free (Visited 15.5.2015).

[54] Unity pro and unity personal software license agreement 5.x (online). URL:
http://unity3d.com/legal/eula (Visited 15.5.2015).

[55] Torque3d (online). URL:
http://www.garagegames.com/products/torque-3d/overview
(Visited 15.5.2015).

A Source code and video

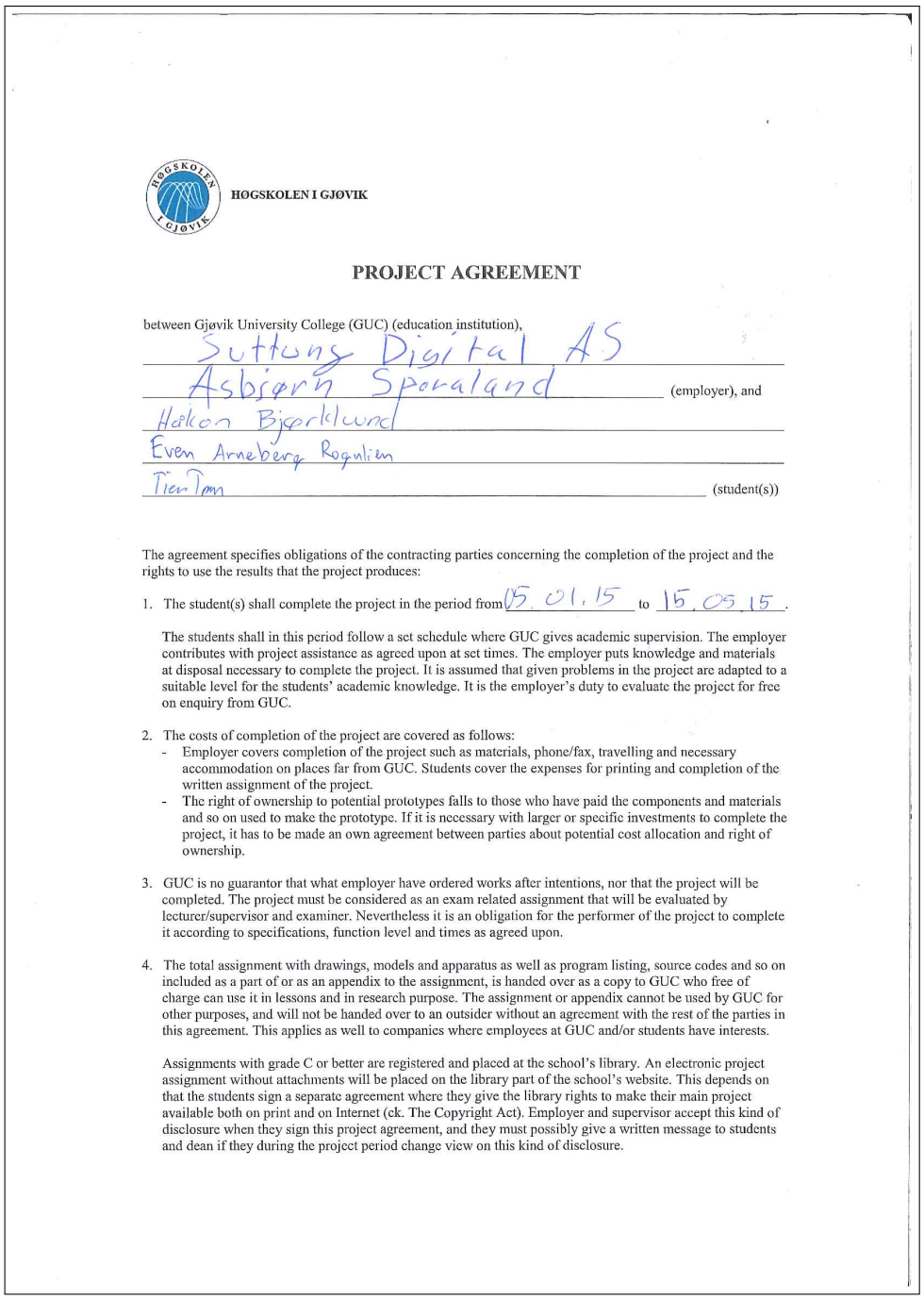
A.1 Source code

The source code can be found on Bitbucket.
The link for the demo/development: https://bitbucket.org/tienqt/noxplus-demo
The link for the engine: https://bitbucket.org/tienqt/noxpluss-engine

A.2 Video

There are some videos made of the final product, they can be found on our website:
http://hovedprosjekter.hig.no/v2015/imt/spill/noxplus/?page_id=43

B Project Agreement



HØGSKOLEN I GJØVIK

PROJECT AGREEMENT

between Gjøvik University College (GUC) (education institution),
Suttung Digital AS
Asbjørn Sporaland (employer), and
Håkon Bjørklund
Even Arneberg Rognlien
Tien Tran (student(s))

The agreement specifies obligations of the contracting parties concerning the completion of the project and the rights to use the results that the project produces:

1. The student(s) shall complete the project in the period from 15.01.15 to 15.05.15.

   The students shall in this period follow a set schedule where GUC gives academic supervision. The employer contributes with project assistance as agreed upon at set times. The employer puts knowledge and materials at disposal necessary to complete the project. It is assumed that given problems in the project are adapted to a suitable level for the students' academic knowledge. It is the employer's duty to evaluate the project for free on enquiry from GUC.

2. The costs of completion of the project are covered as follows:
   - Employer covers completion of the project such as materials, phone/fax, travelling and necessary accommodation on places far from GUC. Students cover the expenses for printing and completion of the written assignment of the project.
   - The right of ownership to potential prototypes falls to those who have paid the components and materials and so on used to make the prototype. If it is necessary with larger or specific investments to complete the project, it has to be made an own agreement between parties about potential cost allocation and right of ownership.

3. GUC is no guarantor that what employer have ordered works after intentions, nor that the project will be completed. The project must be considered as an exam related assignment that will be evaluated by lecturer/supervisor and examiner. Nevertheless it is an obligation for the performer of the project to complete it according to specifications, function level and times as agreed upon.

4. The total assignment with drawings, models and apparatus as well as program listing, source codes and so on included as a part of or as an appendix to the assignment, is handed over as a copy to GUC who free of charge can use it in lessons and in research purpose. The assignment or appendix cannot be used by GUC for other purposes, and will not be handed over to an outsider without an agreement with the rest of the parties in this agreement. This applies as well to companies where employees at GUC and/or students have interests.

   Assignments with grade C or better are registered and placed at the school's library. An electronic project assignment without attachments will be placed on the library part of the school's website. This depends on that the students sign a separate agreement where they give the library rights to make their main project available both on print and on Internet (ck. The Copyright Act). Employer and supervisor accept this kind of disclosure when they sign this project agreement, and they must possibly give a written message to students and dean if they during the project period change view on this kind of disclosure.

5. The assignment's specifications and results can be used by the employer's own work. If the student(s) in its assignment or while working with it, makes a patentable invention, relations between employer and student(s) applies as described in *Act respecting the right to employees' inventions* of 17th of April 1970, §§ 4-10.

6. Beyond the publicising mentioned in item 4, the student(s) have no right to publicise his/hers/theirs assignment, fully or partly or as a part of another work, without consensus from the employer. Equivalent consent must be made between student(s) and lecturer/supervisor regarding the material placed at disposal by the lecturer/supervisor.

7. The students shall hand in the assignment with attachments electronic (PDF) in Fronter. In addition the students shall hand in a copy to the employer.

8. This agreement is drawn up with one copy to each party. On behalf of GUC it is dean/vice dean who approves the agreement.

9. In each case it is possible to enter separate agreement between employer, student(s) and GUC who closer regulate conditions regarding issues such as ownership, further use, confidentiality, cost coverage, and economic utilisation of the results.

   If employer and student(s) wish an additional or new agreement, this will occur without GUC as a party.

10. When GUC also act as employer, GUC accede to the agreement both as education institution and as employer.

11. Possible disagreements concerning understanding of this agreement are solved by negotiations between the parties. If consensus is not achieved, the parties agree that the disagreement is solved by arbitration, according to provision in Civil Procedure Act of 13th of August 1915, no 6, chapter 32.

12. Participants by project implementation:

GUCs supervisor (name): 20.01.2015 Marius Walter

Employers contact person (name): Asbjørn Sporaland

Student(s) (signature): Håkon Bjørklund date 21/01-15

Even A. Rognlien date 21/01-15

Tien Q Tran date 21/01-15

date

Employer (signature): date 21/01-15

IMT Dean/Vice Dean (signature): date 20/4/15

Revised 25th of November, 2010, Hilde Bakke

C Group rules

Group rules for Bachelor project "IMT3912", spring 2015 at HiG

1. A preliminary working schedule is set up the first week of the project for the whole period. It is compulsory attendance on all meetings. Changes in the schedule must happen as soon as possible and at the latest a day before. Skype meetings are allowed. Every member keeps their own log of what work they do and choices they make during this work.

2. The group are to use Bitbucket/Git as a version control. Everyone are to add and update the code, when a module is finished or are done for the day.

3. All group members are to help each other through the whole project duration, with all planned tasks. If a module is delayed the rest of the group shall be notified immediately. The rest of the group is required to help the person responsible for the module with any issues that is slowing the progress. There should be given a time and date for when the delayed module will be made available.

4. Delays who are reported in the same day as the delivery is expected will receive a warning. When three warnings are given or a single delivery is delayed for longer than reasonable, the rest of the group members can send a written message to Supervisor. By repeated warnings and mails the group member can be excluded from the project but this should be a last resort and cannot happen before a meeting with Supervisor. Excluding a member cannot happen later than 14 days before final delivery of the project. This should not happen because of point 3.

5. A plan for systematically testing is to be developed and followed. Testing is compulsory and is just as important as the development and documents. Testing should be done before making the module available. The test may be skipped if the rest of the group members agree.

6. Academic disagreements should be solved as far as possible through constructive discussions and advice from externals, like supervisor, employee or other academic personnel. If the conflict cannot be solved through this a majority vote will conclude the disagreement.

7. Each group member is to spend at the minimum 35 hours per week. If a person cannot work the minimum amount of hours, this person will have to compensate by working more the next week. If a group member is working less a warning will be issued. (See point 4.)
Each group member will log his hours using Toggl.

8. One leader will be appointed with a deputy. The leader will schedule meetings, make sure team members follow up on these and will generally act as a point of communication within the group. In case members are unsatisfied with current leader, they can report this to the deputy who will then make the current leader step down and a new leader is chosen.

9. All code will be commented and clean, and should be readable by all group members. Coding conventions shall also be followed, a link is at the bottom of the document.

Even A. Rognlien
Even Rognlien

Tien Tran
Tien Tran

Håkon Bjørklund
Håkon Bjørklund

17-01-15
Date

Contact information:

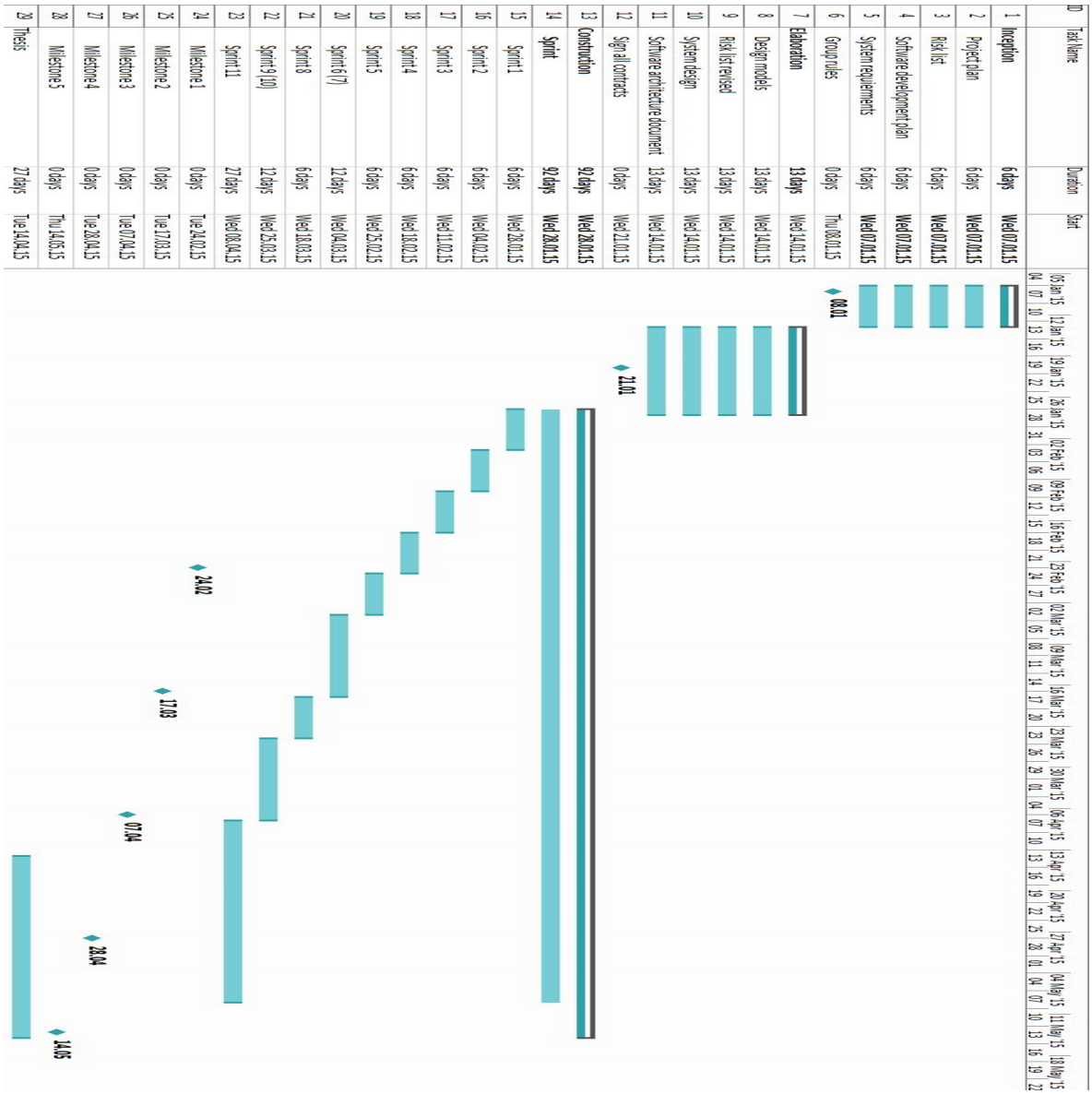
Even A. Rognlien	120649	99528915	even.rognlien@gmail.com
Tien Q. Tran	120647	41387762	tieenqt@gmail.com
Håkon Bjørklund	121059	48062356	hkonbjork@gmail.com

Coding conventions
https://daidata.net/atlassian/confluence/display/NOX/Development+Guidelines

D Risk tabel

Descriptor	Probability	Impact	Risk mitigation plan
Missing group member 7+ days	Moderate	Severe	Missing person works extra hours before and after to make up for it. Also add extra working hours for the rest of the group. Notify team leader as fast as possible and adjust the project plan. Follow group rules.
Internal conflicts	Unlikely	Severe	Follow the group rules.
Physics engine we have integrated cannot do what we wanted it to do.	Rare	Severe	Make small tests to see if things we want is possible before we implement and get to far. Modify the engine to do what we want it do it.
Missing essential equipment	Unlikely	Significant	Buy or borrow equipment. Use the labs if necessary.
Missing milestones	Likely	Moderate	Adjust the project plan or add extra working hours to catch up
Supervisor and Suttung is unavailable for an extended period of time	Unlikely	Moderate	Re-prioritize task list and do what we can without their guidance. Always plan one week ahead, make sure everyone understands their task for the next week.
Delayed access to Nox-engine	Unlikely	Moderate	Start exploring other 3D-engines and write small test modules for Nox-engine if possible. Write a 3D rendering system with little external dependencies, so it is easy to implement in the Nox Engine.
Missing group member. 1 - 7 days	Likely	Minor	Missing person works extra hours before and after to make up for it. Notify team leader as fast as possible and adjust the project plan. Follow group rules.
Problem understanding the Nox-engine	Moderate	Negligible	Arrange a meeting with Suttung or supervisor.

E Gantt chart



F Daily scrum

Monday February 9th
Håkon:
- Create a new camera class for 3D - TrdCamera(). Start on camera movement.

Tien:
- Work with world/scene management; parse world json-files and load 3D actors to the world.

Even:
- Create interface between 3D renderer and Tien's work with world loading.

Tuesday February 10th
Håkon:
- Did:
  - Created a separate camera class
  - Implemented a new camera class only for 3D, to make code more readable
- To do:
  - Finish camera class and basic movement of camera (with mouse look)
  - Create a TransformChange class for 3D
  - Help with scene graph

Even:
- Did:
  - Created interface classes for loading 3D models based on file path from json files.
- To do:
  - Add scene graph for world based on data from json files. Will place static objects in the world.

Tien:
- Did:
  - Created a component class for actor graphics
- Do:
  - Add scene graph for world based on data from json files. Will place static objects in the world.

Wednesday February 11th

Even:

- Did:
  - Together with Tien we implemented our own type of scene graph nodes and managed to load assets and create scene graph dynamically from json file.
- To do:
  - Going on vacation, not much time for work today..

Håkon:

- Did:
  - Almost finished camera. Camera moves with keys and mouse. Wanted to remove the windows lag when pushing buttons.
- To do:
  - Finish that damn camera!
  - Create a TransformChange class for 3D
  - Clean up
  - Help Tien

Tien:

- Did:
  - Add scene graph for world based on data from json files. Will place static objects in the world.
- To Do:
  - Clean up.
  - Update scenegraph with new transform matrices.

Thursday February 12th

Tien:

- Did:
  - Cleaned up
- To Do:
  - Integrate Bullet
  - Test TrdTransformChange class

Håkon:

- Did:
  - Moved camera controls out of camera and into TrdApplication.

        - Created TrdTransformChange class.
        - Cleaned up
    - To Do:
        - Integrate Bullet
        - Test TrdTransformChange

Friday February 13th

Tien:
- Did: Tried to add bullet as a submodule for the nox-engine, but ran into internal linking problems within bullet cmake.
    -
- To Do:
    - Solve bullet internal link problems.

Håkon:
- Did:
    - Tried to add bullet as a submodule.
    - Finished TrdTransfromChange

- To Do:
    - Learn Bullet
    - Solve bullet internal link problems.

Saturday February 14th

Tien:
- Did:
    - Solved bullet internal linking problems and added bullet as a subproject for the nox-engine.
- To Do:
    - Read bullet documentation.

Håkon:
- Did:
    - Looked at Bullet tried to learn how it works.
    - Tried finding the linking problem.

- To Do:
    - Learn Bullet

Sunday February 15th

Tien:

- Did:
  - Read bullet documentation
  - Study hello world bullet application.
- To Do:
  - Set up empty skeleton class for bullet intergration with nox.

Håkon:

- Did:
  - Read documentation and setup Bullet in it's own project.
  - Ran some debugging
  - Pulled in pull request from Magnus

- To Do:
  - Create skeleton classes for bullet integration with Nox.

Monday February 16th

Tien:

- Did:
  - Created a simulation class and interface class for bullet.
  - Added bullet simulation class into nox system.
  - Discarded everything
- To Do:
  - Read about syncing bullet with rendering system.

Håkon:

- Did:
  - Created skeleton classes for Bullet integration. (Pushed)
  - Started testing out implementation.

- To Do:
  - nothing (Artificial Intelligence presentation on Tuesday)

Tuesday February 17th

Tien:

- Did:
  - Read bullet documentation.
- To Do:
  - Create a test branch for bullet.

        - Create a prototype for bullet simulation class.

Håkon:
- Did:
    - Nothing, AI.

- To Do:
    - Try setting up init function and addRigidBody function for Bullet

Wednesday February 18th

Tien:
- Did:
    - Created a test class for bullet simulation:
        - Init and loading physics object from json files with basic colision shapes.
- To Do:
    - Clean up bullet-test branch.
    - Close current sprint and set up a new sprint.

Håkon:
- Did:
    - Sat up the init function. Tried creating the rest. Failed. Discarded

- To Do:
    - Study Bullet some more.
    - Fixed the backlog and sprint.

Thursday February 19th

Tien:
- Did:
    - Close current sprint and started a new sprint.
    - Cleaned up bullet test branch.
- To Do:
    - Sync bullet with rendering system using a custom motion state class.

Håkon:
- Did:
    - Looked at Bullet some more.

- To Do:
    - Create function to create different shapes

Friday February 20th

Tien:
- Did:
  - Syncing bullet using custom motion state class.
- To Do:
  - Create concave hull shape.

Håkon:
- Did:
  - Created function for different collision shapes
- To Do:
  - Look at Rotation and callback

Even:
- Did:
  - Nothing
- ToDo:
  - Start looking at what Tien and Håkon have done and look at concave hull shapes for Bullet physics

Saturday February 21th

Tien:
- Did:
  - Nothing.
- To Do:
  - Vietnamese new year celebration and birthday party.
  - Create the Directory structure with the group.

Håkon:
- Did:
  - .Nothing
- To Do:
  - Do what i was supposed to do.
  - Create the Directory structure with the group.

Even:

- Did:
    - Looked at Tien and Håkons code.
- ToDo:
    - Create the Directory structure with the group.
    - Look more at Tien and Håkons code while looking at concave hull shapes

Sunday February 22th

Tien:
- Did:
    - Create the Directory structure with the group.
- To Do:
    - Refactor current rendering system to use RenderData.
    - Implement debug render mode.
    - Fix documentation.

Håkon:
- Did:
    - Create the Directory structure with the group.
    - Tried fixing the rotation with quaternions, between Bullet and TrdTransform. Need some help, asked for meeting with Nowostawski.

- To Do:
    - Create the callback system.
    - Fix documentation.

Even:
- Did:
    - Create the Directory structure with the group. Started looking at concave hull shapes.
- ToDo:
    - Look more at Tien and Håkons code while looking at concave hull shapes
    - Fix documentation.

Monday February 23th

Tien:

- Did:
  - Fixed documentation.
  - Looked at renderings system.
- To Do:
  - Implement debug rendering.
  - Refactor rendering system to use RenderData.

Håkon:

- Did:
  - Fixed documentation
  - Tried fixing the rotation and implement the callback
- To Do:
  - Ask for some help.
  - Fix the rotation/ create readin functions that do rotation/scale/translation
  - Implement the callback properly.
  - Look at Mathematics for 3D Game Programming and Computer Graphics
  - Look at Game Coding Complete

Even:

- Did:
  - Looked at overall progress vs remaining work, together with group.
  - Tried to find a way of getting mesh data from BulletSimulation class.
- ToDo:
  - Look more at how to access mesh data from inside BulletSimulation class.

Tuesday February 24th

Tien:

- Did:
  - Read about different rendering system.
- To Do:
  - Implement debug renderer using renderData.

Håkon:

- Did:
  - Asked for help. Almost no help received when asked.
  - Looked at Mathematics for 3D Game Programming and Computer Graphics
  - Looked at Game Coding Complete
  - Started implementing callback

        - fixed rotation

- To Do:
    - Finish callback

Even:

- Did:
    - Looked more at how to access mesh data from inside BulletSimulation class.
    - Started implementing Bullet debug rendering together with Tien
- ToDo:
    - Finish debug rendering
    - Start looking at textures.

Wednesday February 25th

Tien:

- Did:
    - Implemented debug renderer with Even.
- To Do:
    - Read about world management (Game coding complete).
    - Read about bitbucket
    - Close branch

Håkon:

- Did:
    - Finished callback

- To Do:
    - Create the event system for the callback
    - Clean up rotation mess
    - Read about time manipulation

Even:

- Did:
    - Finished debug rendering
    - Discussed code and stuff
- ToDo:
    - Finish sprint and setup new with group
    - Look at textures

Thursday February 26th

Tien:
- Did:
  - Read about world management (Game coding complete).
  - Read about bitbucket
  - Closed branch
- To Do:
  - Implement scene graph

Håkon:
- Did:
  - Read about time manipulation
  - Wrote some documentation for sprints.

- To Do:
  - Finish the event system for callback

Even:
- Did:
  - Added loading of textures
- ToDo:
  - Optimize loading, management and switching of textures
  - Fix some texture issues

Friday February 27th

Tien:
- Did:
  - Created a class overview for our current system.
    Read about scene graph.
- To Do:
  - Refactored scene graph with even.
  - Look at alpha rendering

Håkon:
- Did:
  - Finished the event system for callback

- To Do:
  - Clean up personal log and callback.

Even:

- Did:
  - Created class overview with the rest of the group, mapping our current system.
  - Optimized switching of textures using the RenderData class
- ToDo:
  - Improve scene graph
  - Look at rendering steps/passes/alpha textures

Saturday February 28th

Tien:

- Did:
  - Refactored scene graph with even.
  - Implemented alpha rendering with even.
- To Do:
  - fix alpha rendering issue.

Håkon:

- Did:
  - Cleaned up personal log and callback
- To Do:
  - Weekend off

Even:

- Did:
  - Refactored scene graph with Tien.
  - Implemented two render passes, one for solid meshes and one for alpha meshes.
- ToDo:
  - Fix alpha issue

Sunday March 01th

Tien:

- Did:
  - Didn't manage to fix alpha rendering issue.
- To Do:
  - Fixed rotation problems in bulletSimulation/actorMotionState.
  - Create convexhullshape with even.

Even:

- Did:
  -
- ToDo:
  - Look at rotation problem with Tien

Monday March 02th

Tien:

- Did:
  - Fixed rotation problems in bulletSimulation/actorMotionState.
  - Create convexhullshape with even.
- To Do:
  - Update class overview
  - Memory leak.
  - Bullet rotation.

Håkon:

- To Do:
  - Finish class diagram with group
  - Merging changes to Nox and pull requests with group
  - Try to do onCollision differently.

Even:

- Did:
  - Added support for loading convex and concave hull shape into Bullet simulation.
  - Changed debug renderer to use Bullets own debug colors.
- ToDo:
  - Refactor RenderSdlWindowView to support both 3D and 2D rendering

Tuesday March 03th

Tien:

- Did:
  - Update class overview.
  - Memory leak in event system.
  -
- To Do:
  - Merged pull requests from suttung.

- Synced nox fork with suttung.

Håkon:
- Did:
  - Finish class diagram
  - Merging done
  - Still trying to do onCollision differently from event queue system

- To Do:
  - Keep trying to implement the onCollision

Even:
- Did:
  - Merged TrdRenderSdlWindowView with RenderSdlWindowView to make support for both 3D and 2D rendering.
  - Update class diagrams with group.
- ToDo:
  - Look at memory leaks.
  - Look at Tien merging pull requests from Suttung

Wednesday March 04th

Tien:
- Did:
  - Write missing documents
- To Do:
  - Set up new sprint.
  - Look at deferred rendering.

Håkon:
- Did:
  - Failed implementing the onCollision differently

- To Do:
  - Look at the onCollision from developers viewpoint. Work with the group.
  - Implement new collision Shapes.

Even:
- Did:
  - Looked at memory leaks.
  - Watched Tien merging pull requests from Suttung

- ToDo:
    - Do sprint planning with group.
    - Look at deferred rendering

Thursday March 05th

Tien:
- Did:
    - Tried basic deferred rendering.
- To Do:
    - Setting up Api for collision.
    - Read about frame buffers.
    - Try deferred lighting.

Håkon:
- Did:
    - Logs and meetings
    - read up on new collision shape

- To Do:
    - Implement new collision shape

Even:
- Did:
    - Sprint planning
    - Started reading about deferred rendering
- ToDo:
    - Fix compound collision shapes with Håkon

Friday March 06th

Tien:
- Did:
    - Had problems with rendering light.
    - Forgot to activate the textures before usage.
- To Do:
    - Render direction and point light.

Håkon:

- Did:
    - Implemented Compound shape

- To Do:
    - Nothing before tuesday, AI assignment.

Even:
- Did:
    - Fixed compound collision shapes with Håkon
- ToDo:
    - Work with deferred rendering with Tien

Saturday March 07th

Tien:
- Did:
    - Implemented directional and point light. Error when looking thru and entering light volume.
- To Do:
    - Fix the problems above using stencil buffer.

Even:
- Did:
    - Worked with Tien on deferred rendering
- ToDo:
    - day off

Sunday March 08th

Tien:
- Did:
    - Read about stencil buffers.
- To Do:
    - Fix lighting problems using stencil buffer.

Even:
- Did:
    - nothing
- ToDo:
    - Work more on deferred rendering

Monday March 09th

Tien:

- Did:
  - Fixed the lighting issue using stencil buffer.
- To Do:
  - Implement spot light.

Even:

- Did:
  - Worked with Tien on deferred rendering
- ToDo:
  - Work more on deferred rendering

Tuesday March 10th

Tien:

- Did:
  - Implemented spotlight with even.
- To Do:
  - Read about different ways to implement time traversing in game.

Håkon:

- To Do:
  - Clean up rotation
  - Check error on laptop
  - Meeting with the rest

Even:

- Did:
  - Worked with Tien on deferred rendering
  - Started playing with time rewinding
- ToDo:
  - Study time rewinding in the game Braid,
  - Look more at time rewinding
  - Finish deferred rendering, making it generic (not hard-coded)

Wednesday March 11th

Tien:

- Did:
  - Discussed different ways to implement time traversal with the group.
- To Do:
  - Close current sprint.
  - Set up new sprint.
  - Write sprint/meeting documents.

Håkon:

- Did:
  - cleaned up rotation
  - checked error on laptop
  - had meeting with the rest of the group.
- To Do:
  - Meetings
  - Logs

Even:

- Did:
  - Looked at video about the time management in the Braid game
  - Discussed different ways of storing time history with group
- ToDo:
  - Retrospective meeting
  - Set up new sprint

Thursday March 12th

Tien:

- Did:
  - Write meeting/logg documents.
- To Do:
  - look at nox control system.

Håkon:

- Did:
  - Meetings
  - Logs
- To Do:
  - Implement Collision Detection API

Even:

- Did:
    - Meetings
- ToDo:
    - Look at Assimp animations/bone structure.

Friday March 13th

Tien:

- Did:
    - Looked at nox control system.
- To Do:
    - Look at raytest used in bullet.

Håkon:

- Did:
    - Implementing Collision Detection API
- To Do:
    - Continue with Collision Detection API

Even:

- Did:
    - Started looking at assimp's animation system. Decided to rewrite our mesh system.
- ToDo:
    - Rewrite mesh loading to work better with animations.

Saturday March 14th

Tien:

- Did:
    - Look at raytest used in bullet.

Håkon:

- Did:
    - Continued with Collision Detection API
- To Do:
    - Private matters to attend.

Even:

- Did:
    - Started rewriting mesh loading (TrdSceneLoader)
- ToDo:
    - Weekend

Sunday March 15th

Tien:

- To Do:
    - Implement mouse picking using raytest.

Håkon:

- To Do:
    - Visitors

Even:

- Did:
    - Weekend
- ToDo:
    - Weekend

Monday March 16th

Tien:

- Did:
    - Implement mouse picking using raytest.
- To Do:
    - Try to move selected actor towards mouse cursor.

Håkon:

- To Do:
    - Continue with Collision Detection API

Even:

- Did:
    - Weekend
- ToDo:
    - Continue on new mesh loading

Tuesday March 17th

Tien:
- Did:
  - Tried to move selected actor towards mouse cursor.
- To Do:
  - Implement actorcontrol for 3dDirection.

Håkon:
- Did:
  - Continued with Collision Detection API:
- To Do:
  - Finish Collision Detection API:
  - Look at the last shapes for collision shapes.

Even:
- Did:
  - Worked on mesh loading.
- ToDo:
  - Start working on animations again

Wednesday March 18th

Tien:
- Did:
  - Started on actor control for 3dDirection.
- To Do:
  - Finish actor control for 3dDirection.

Håkon:
- Did:
  - Looked at reusing shapes for collision saving memory.
- To Do:
  - Finish Collision Detection API.
  - Look at the last shapes for collision shapes.

Even:
- Did:
  - Did some work on animations
- ToDo:
  - Finish animations

Thursday March 19th

Tien:
- Did:
  - Modified Noxs control system to support 3d
  - Helped even with animation problem; bones are stretched all over the place. did not find the problem.
- To Do:
  - Start on the directional control component.
  - Help Even with solving the bone problem.

Håkon:
- Did:
  - Finished Collision Detection API
- To Do:
  - Look into BVHTriangle collision shape

Even:
- Did:
  - Finished animation class
- ToDo:
  - Find out why the animations look so deformed, and fix it.

Friday March 20th

Tien:
- Did:
  - Started on the directional control component.
  - Found the problem with the deformed bones.
- To Do:
  - Finish the directional control component.

Håkon:
- Did:
  - Looked into BVHTriangle collision shape.
- To Do:
  - Look into deformable mesh/collision shape

Even:
- Did:
  - With great help from Tien, we fixed the deformed animations.
- ToDo:
  - Create game developer API for controlling the animations.

Saturday March 21th

Tien:
- Did:
  - Almost finished the directional control component.
  - Tried to fix onUpdate function not called for components
- To Do:
  - Finish directional control component

Håkon:
- Did:
  - Started with deformable mesh
- To Do:
  - Help Tien with Actor Movement
  - Help Even with Animation
  - Look into reuse of collision shapes

Even:
- Did:
  - Started on animation control API.
- ToDo:
  - Look at animation optimization (threading?)

Sunday March 22th

Tien:
- Did:
  - Finished the directional control component.
- To Do:
  - N/A

Håkon:
- Did:
  - Helped Tien with Actor Movement
  - Helped Even with Animation
  - Looked into reuse of collision shapes

Even:
- Did:
  - Looked at threading together with Håkon.

Monday March 23th

Tien:
- Did:
  - N/A
- To Do:
  - Look at memory leak
  - Finish rotational control component

Håkon:
- To Do:
  - Look into threading
  - Help Even with Animation
  - Help Tien with Actor Controls
  - Continue with Deformable mesh

Even:
- ToDo:
  - Look more at threading with Håkon; how and where to do it.

Tuesday March 24th

Tien:
- Did:
  - Finished rotational control component.
  - Fixed memory leak
  - Fixed component not being updated problem.
- To Do:
  - Read about shadow mapping
  - Clean up
  - Map time manipulation classes.

Håkon:
- Did:
  - Looked into threading the renderer.
  - Helped Even with Animation
  - Helped Tien with Actor Controls
  - Continued with Deformable mesh
- To Do:
  - Refactor Collision detection/ API
  - Testing Impact mesh

Even:

- Did:
    - Looked more at and discussed threading with Håkon. Decided to talk with Mariusz about it.
- ToDo:
    - Create interface for updating and re-buffering vertex positions for a mesh (Håkon will use it for impact (deformable) collision shapes).

Wednesday March 25th

Tien:

- Did:
    - Read about stencil shadow mapping.
    - Mapped time manipulation classes
- To Do:
    - Rename all classes.

Håkon:

- Did:
    - Refactored Collision detection / API
    - Tested impact Mesh, don't work.
- To Do:
    - Meetings/Documents
    - Look more into threading the renderer. Cleaning up the includes, commenting.

Even:

- Did:
    - Created interface in TrdMesh for updating and re-buffering vertex positions.
- ToDo:
    - Meetings.

Thursday March 26th

Tien:

- Did:
    - Renamed all classes fixing the sufix
- To Do:
    - Sync nox with noxplus.

Håkon:

- Did:
    - Cleaned up includes
- To Do:
    - Implement reuse of collision shapes.

- Tweak collision shapes
- Fix warnings.
- Animation optimisation with Even
- Help Tien with Pulling/merging latest Nox engine update.

Even:
- Did:
  - Meetings + Tried to help Tien with pulling/merging latest NOX engine.
- ToDo:
  - Work with another course..

Friday March 27th

Tien:
- Did:
  - Failed to sync nox with noxplus. Problems build new third party lib added by nox.
- To Do:
  - Try to fix syncing problem.

Håkon:
- Did:
  - Fixed warnings.
  - Implemented Reuse of collision shapes for convex shapes
  - Helped Even with animation optimisation.
  - Tweaked collision shape implementation.
- To Do:
  - Artificial Intelligence course

Even:
- ToDo:
  - Fix performance issue with multiple animations: generate animation “frames” before game starts.

Saturday March 28th

Tien:
- Did:
  - Suttung fixed the third party lib.
  - Failed to sync again. The third party lib won’t link properly.
- To Do:
  - Fix link problem with the third party lib.

Even:

- Did:

Sunday March 29th

Tien:

- Did:
    - Tried to fix linker problem by going thru CMake files.
- To Do:
    - Light

Even:

- Did:
    - Friday: Started implementing animation optimization. Huge performance boost!
- ToDo:
    - Fix independent animations for actors with same mesh.

Monday March 30th

Tien:

- Did:
    - Found the problem with linking the third party lib.
    - Started on lighting
- To Do:
    - Read oglDev deferred shading tutorial

Håkon:

- To Do:
    - Play around with time manipulation.

Even:

- Did:
    - Started working on making animations independent of mesh. Put animationdata into “ActorGraphics3d” so every actor has their own.
- ToDo:
    - Finish and clean up new animation structure.

Tuesday March 31th

Tien:

- Did:
    - Read all three part of oglDev tutorial
- To Do:
    - Light

Håkon:

- Did:

        - Tried some simple rewinding.
    - To Do:
        - Easter (The Gathering)

Even:
- Did:
    - Finished, cleaned up and pushed the new animation structure.
- ToDo:
    - Write log for last two weeks.
    -

Wednesday April 01th

Tien:
- Did:
    - Light
- To Do:
    - Light

Even:
- Did:
    -
- ToDo:

Thursday April 02th

Tien:
- Did:
    - Light
- To Do:
    - Light

Even:
- Did:
    - Cleanup Texture3d and OpenGlRenderer3d
- ToDo:
    - Look at lighting with Tien.

Friday April 03th

Tien:
- Did:
    - Light
- To Do:
    - Light

Even:
- Did:
    - Looked at lights with Tien.
- ToDo:

        - Look at kinematic bodies.

Saturday April 04th

Tien:

- Did:
    - Light.
    - Render to texture.
- To Do:
    - Light

Even:

- Did:
    - Looked at kinematic bodies.
- ToDo:
    - Look at kinematic bodies

Sunday April 05th

Tien:

- Did:
    - Light
- To Do:
    - Light

Even:

- Did:
    -
- ToDo:
    -

Monday April 06th

Tien:

- Did:
    - Light
- To Do:
    - Set up classes for time manipulation.

Even:

- Did:
    - Light with Tien.
- ToDo:
    - Set up classes for time manipulation.

Tuesday April 07th

Tien:

- Did:
    - Time manipulation
- To Do:

        - Time manipulation

Håkon:

- To Do:
    - Work with Tien and Even on Time manipulation.

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Wednesday April 08th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Worked with Tien and Even on time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Thursday April 09th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Work with Tien and Even on Time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:

        - Time manipulation
- ToDo:
    - Time manipulation

Friday April 10th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Work with Tien and Even on Time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Saturday April 11th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Work with Tien and Even on Time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Sunday April 12th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Work with Tien and Even on Time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Monday April 13th

Tien:

- Did:
    - Time manipulation
- To Do:
    - Time manipulation

Håkon:

- Did:
    - Work with Tien and Even on Time manipulation.
- To Do:
    - Continue with time manipulation

Even:

- Did:
    - Time manipulation
- ToDo:
    - Time manipulation

Tuesday April 14th

Tien:

- Did:
  - Time manipulation
- To Do:
  - Time manipulation

Håkon:

- Did:
  - Work with Tien and Even on Time manipulation.
- To Do:
  - Setup thesis and start writing.

Even:

- Did:
  - Time manipulation
- ToDo:
  - Time manipulation

Wednesday April 15th

Tien:

- Did:
  - Time manipulation
- To Do:
  - Time manipulation

Håkon:

- Did:
  - Sat up thesis and started writing Introduction.
- To Do:
  - Continue with thesis.

Even:

- Did:
  - Time manipulation
- ToDo:
  - Time manipulation

Thursday April 16th

Tien:

- Did:
  - Time manipulation
- To Do:
  - Time manipulation
  - Thesis

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - Time manipulation
- ToDo:
  - Time manipulation
  - Thesis

Friday April 17th

Tien:

- Did:
  - Time manipulation
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - Time manipulation
  - Thesis
- ToDo:
  - Thesis

Saturday April 18th

Tien:

- Did:
    - Time manipulation
    - Thesis
- To Do:
    - Time manipulation
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    -
- ToDo:

Sunday April 19th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Artificial Intelligence project

Even:

- Did:
    - Thesis
- ToDo:
    - Thesis

Monday April 20th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Even:

- Did:
  - Thesis
- ToDo:
  - Thesis

Tuesday April 21th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.
  - Webpage

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - Thesis
- ToDo:
  - Thesis

Wednesday April 22th

Tien:

- Did:
  - webpage
  - thesis.
- To Do:
  - Thesis.
  - Webpage.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - Thesis
- ToDo:
    - Thesis

Thursday April 23th

Tien:

- Did:
    - Thesis
    - webpage.
- To Do:
    - Thesis

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - Writing on requirement specifications
- ToDo:
    - Writing on requirement specifications

Friday April 24th

Tien:

- Did:
    - Thesis
- To Do:
    - Create class diagrams.

Håkon:

- Did:
    - Writing thesis
- To Do:

        - Continue with thesis.

Even:
- Did:
    - Writing on requirement specifications
- ToDo:
    - class diagram with Håkon

Saturday April 25th
Tien:
- Did:
    - Created class diagram.
- To Do:
    - Thesis

Håkon:
- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:
- Did:
    - class diagram with Håkon
- ToDo:
    - system sequence diagram with Tien

Sunday April 26th
Tien:
- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:
- Did:
    - Writing thesis
- To Do:
    - Start on demo

Even:

- Did:
    - system sequence diagram with Tien
- ToDo:
    - writing on design

Monday April 27th

Tien:

- Did:
    - Thesis.
- To Do:
    - Logging of component.
    - Setting up demo.

Håkon:

- Did:
    - Started on demo with even.
- To Do:
    - Continue with thesis.

Even:

- Did:
    - Started on demo with Håkon. Fixed issue with non-sleeping physics shapes.
- ToDo:
    - Finish demo. Look at possible bug with collision callbacks (?). Crash when more than one callback in world ??

Tuesday April 28th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - Worked on Demo
- ToDo:

        - writing on design
        - looked at how to save user created components, with Tien

Wednesday April 29th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - looked at how to save user created components, with Tien
- ToDo:
    - rewriting some parts based on feedback from Mariusz. With Håkon

Thursday April 30th

Tien:

- Did:
    - Thesis.
- To Do:
    - gTest.
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - rewriting some parts based on feedback from Mariusz. With Håkon
- ToDo:
    - rewriting some parts based on feedback from Mariusz. With Håkon

Friday May 1th

Tien:

- Did:
  - gTest.
  - Thesis.
- To Do:
  - gTest

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - rewriting some parts based on feedback from Mariusz. With Håkon
- ToDo:
  - rewriting some parts based on feedback from Mariusz. With Håkon and Tien

Saturday May 2th

Tien:

- Did:
  - gtest
- To Do:
  - N/A

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - rewriting some parts based on feedback from Mariusz. With Håkon and Tien
- ToDo:
  - rewriting some parts based on feedback from Mariusz. With Håkon and Tien

Sunday May 3th

Tien:

- Did:
  -
- To Do:
  - Refactor time manipulation.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Work with demo

Even:

- Did:
  - rewriting some parts based on feedback from Mariusz. With Håkon and Tien
- ToDo:
  - worked on demo. Conflict solver (interruption solving) + håkon: skybox

Monday May 4th

Tien:

- Did:
  - Refactored time manipulation.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Created 3D models with texture, skybox and plane.
  - Wrote a bit on thesis.
- To Do:
  - Continue with thesis.

Even:

- Did:
  - worked on demo. Conflict solver (interruption solving) + håkon: skybox
- ToDo:
  - write on implementation

Tuesday May 5th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - write on implementation
- ToDo:
    - write on implementation

Wednesday May 6th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.
    - Write logs and fix documents.

Even:

- Did:
    - write on implementation
- ToDo:
    - write on implementation

Thursday May 7th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
  - fixed some logs
- To Do:
  - Continue with thesis.

Even:

- Did:
  - write on implementation
- ToDo:
  -

Friday May 8th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - write on implementation/discussion
- ToDo:
  - write on discussion

Saturday May 9th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - write on discussion
- ToDo:
    - write on discussion

Sunday May 10th

Tien:

- Did:
    - Thesis.
- To Do:
    - Thesis.

Håkon:

- Did:
    - Writing thesis
- To Do:
    - Continue with thesis.

Even:

- Did:
    - write on discussion
- ToDo:
    - write on discussion

Monday May 11th

Tien:

- Did:
  - Thesis.
- To Do:
  - Logging og components.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - write on discussion
- ToDo:
  - write on discussion

Tuesday May 12th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - write on discussion
- ToDo:
  - rewrite on suggestions from Mariusz+ some performance testing

Wednesday May 13th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - rewrite on suggestions from Mariusz+ some performance testing
- ToDo:
  - performance testing + polishing requirements

Thursday May 14th

Tien:

- Did:
  - Thesis.
- To Do:
  - Thesis.

Håkon:

- Did:
  - Writing thesis
- To Do:
  - Continue with thesis.

Even:

- Did:
  - performance testing + polishing requirements
- ToDo:
  - polishing thesis

G Milestone review

Review of Milestone 1, 04.02.15 - 24.02.15:
What we have:

- Simple scene (Basic rendering system)
- Basic shaders (Position transformation and fixed color)
- Basic modelloader (Loading Vertex, normal, uv and material.)
- Basic physics (Basic collision shapes and simple collision callback)
- Demo of physics

Time spent: 3 sprints. 04.02.15 - 24.02.15
Time left: 10 sprints.
Next Milestone review:

what we expect for the next 3 sprints:

- Advance rendering system (threading/deferred rendering)
- Advance modelloader (Texture support, bone/joint animation)
- Advance physics (Advance collision shape, soft body mesh collision)
- Basic to advanced time manipulation (Basic: rewind controlled actor position. Advanced: Rewind every actor position).
- Website

Review of Milestone 2, 25.02.15 - 17.03.15:
What new we have:

- Advanced physics(advanced collision shapes and a proper callback system)
- Modelloader (Support for bones and texture)
- Advanced rendering system (deferred rendering and alpha rendering)
- Light (point, directional and spotlight)
- Actor controls
- Animation

Different tasks took longer than expected like Physics and Animation. We are currently looking into threading either the renderer or just the calculations of animation bone data. Time manipulation was pushed back to next sprint so we could finish up most of the tasks we felt was necessary for a good starting base. Soft body mesh is down prioritized to the point that it might not be implemented at all, the same will happen with some of the most complex collision shapes. We were told that the website is not very important either so we down prioritized that too.

What we expect for the next 3 sprints:

- Threaded renderer
- Demos (at least 4, one for rendering, one for physics, one for light, one for animation)
- Advanced time manipulation
- Shadow mapping but it has very low priority.
- Clean up, comment and verify that we have followed Suttung’s coding conventions.

- Get review/feedback on our code from Mariusz

Review of Milestone 3, 18.03.15 - 07.04.15:
What new we have:

- Optimized animation rendering
- API for animation control
- Physics optimization
- Lighting component
- Better actor control
- An early overview of how we want the time manipulation API

We looked at how we could make the rendering run on another thread. The problem was that OpenGL must be initialized on the same thread as it is going to be used. Anyways, the main bottleneck was the calculations of bone transformations for each frame, so we got a tip from Mariusz that we could just calculate all the frames once and fetch them when they need to be used. This was very efficient.

Also the threading of the renderer would take too much time since there would be much that had to be changed. Some changes could also ruin the 2D part of the engine.
Advanced time manipulation.
We forgot about easter, and most of us had plans and little was done in that time period.
We haven't had time for getting some feedback from Mariusz.
Shadow mapping is dropped.
Still need some cleaning and commenting, but we are soon done.

What we expect for the next 3 sprints

- Thesis started (60% done)
- Time manipulation
- Extended physics API
- Website
- Finish cleaning and commenting

Review of Milestone 4, 08.04.15 - 28.04.15:
What new we have:

- Time manipulation (still need some tweaking)
- Thesis 33.3...3% done
- Extended physics API
- Website

The fundamentals of our time manipulation is done. It took some days of trying and failing before we found a working solution. We have started on a conflict solver that the game

developer can set to decide different time manipulation paradoxes. We have not implemented the detection of conflicts, but this will be done. A demo is under development - currently there is just a wall of boxes and a camera that will shoot boxes in the direction it is looking. The current version can be used for demonstrating basic rewind and replay. We need to add more example scenarios in the demo.

We started writing slowly on april 17th. After 12 days we have written about 40 pages. It will probably will be some minor and major adjustments during the rest of the writing process, and we believe we have about ⅓ of the thesis done by now.

Website is up, no content, but we will put something from our thesis in there. There were no commenting or cleaning done this week so we still have that left. It should not take that long however. The extension on the physics API took a day, it was brain dead work.

What we expect for the next 2 weeks + 2 days:

- Time manipulation tweaked.
- Demo
- Thesis finished
- Finished cleaning and commenting.

Review of Milestone 5, 29.04.15 - 14.05.15:
What new we have:

- Demo 100% done.
- Thesis 100% done.
- Conflict detection.
- Refactor world logger to use list.
- Complete webpage.
- Finished cleaning and commenting.

We finished writing the thesis, it took quite a long time. And it needed a lot of refactoring. Still could have been better, but everything can always be better.

The demo environment was changed some, and additional models was loaded and added, so we have more to play with.

We refactored the world logger and implemented conflict solving and logging of components. The world logger is now using list instead of arrays and the indexes has been cleaned up making the code more readable. The conflict solver can now detect conflicts and let the game developer handle them. Components can now be logged and rewinded/replayed.

1 Milestone:
Simple scene
Basic shaders
Basic modelloader
Basic physics
Demo of physics

2 Milestone
Advance rendering system (threading/deferred rendering)
Advance modelloader
Advance physics
Basic to advanced time manipulation.
New demo of time manipulation

3 Milestone
Advanced time manipulation.
Optimization
Finish off things that we haven't had time for.
New demo of everything.

4 Milestone
Thesis start 60% finished
Website

5 Milestone
Thesis is main focus

H Meetings

13.1 Meeting with Suttung and Nowostawski

- Time, physics and 3D-rendering as one module
- Go very simple first when starting the new module for the engine and then dig down and go deep.
- Will we use off-the-shelf physics engine or build our own?
    - Need to look at rolling backward and forward physics
    - inverse kinematic
- Find examples of games using timeline manipulation
    - braid
- Physics engine examples:
    - Havoc
    - Bullet
- Suttung would like 3D module for the engine but is also interested in the timeline module.
- Timeline module is selling point for us.
- Need to find the limitations for the different engines we are considering, what is possible and what is not.
- How will we do the timeline?
    - Record events, replay it
    - resimulation
- Effect is the most important thing.

14.1 Meeting with Nowostawski

- Plan milestones
- Technologies
    - We are spreading too thin, we need to focus more on our part and not the Engine itself.
    - Find our core dependencies
    - We should not for example be dependent on SDL.
    - Assetloader
    - OpenGL
    - etc.
- Find out how dependent our modules are on NOX.
- Mariusz believes we should use Bullet as physics engine
    - fastest
    - does a lot on the gpu already
    - supports opencl
- Take a look at Havoc too, but it is pay to use.
- Have a Future Work section
    - time manipulation for 2D

- Stress more that the engine is to be open source
    - add it to project plan where it is applicable
    - simplify core library for other students
- Group rules need change
    - No warning
    - How are we going to work
    - stress positiv more
    - last resort to kick people
- How much will the two groups interact? and how?
- Find different games that use time manipulation
    - How is it used?
    - Try out different ways, experiment.
    - reviews of existing games
        - play or check out gameplay and review them.
- Create a simple world with player first for the demo
- Define other roles as well, infrastructure, assets, integration with NOX, version management etc.
- Can use the background from the assignments Simon wrote

14.1 Meeting with Suttung

- Receive the engine later today
- They are using an MIT license since they find it easier to understand than the open source license.
- Remove section of ownership from the contract
- Signing contract next wednesday
- Engine:
    - Multiple threads for:
        - preparing things that are sent to the gfx card
        - AI
    - Actor is things that have an ID, consists of components.
    - The functionality is in the components.
    - The World handler handles the actors.
    - Event manager maintains the communication between systems. Example: transformations
    - Gameview from the document is View in the engine.
    - The player and AI send them same logic.
- A lot of talk about Suttung, it was nice hearing about their company.

21.01 Meeting with Nowostawski

- Since it's a large and difficult thing we are doing, Mariusz means we should narrow our idea.
- Company of Myself is a game that plays on time manipulation
- Another game, cursor time? tenth ..
- We should figure out the demo and use it as a goal. It should have a simple environment.
- When figuring out the demo we should choose a "time mechanic" we want to implement
- Mariusz thinks it could be beneficial with a mailing list, so if we have questions we can use that to contact Suttung or the other bachelor group working with NOX.
- When we mentioned that we are having abit problems with understanding NOX, Mariusz said we should try for a couple of weeks if we still are having problems we need to figure something out.
- Project plan can be changed after the due date
- Give Mariusz a message when we made changes that we want him to look at. (documents and such),
- We can give him a message this friday to have him look at the project plan since it's due next week.

21.01 Meeting with Suttung

- They believe their code is self explanatory, but they agree that certain parts are not very well documented.
- They are prioritizing writing an overview of the different classes.
- We might have to find some sort of Texture atlas, They are making one but we might need our own.
- Project agreement is signed
- They tried showing us some more code.

28.01 Meeting with Nowostawski postponed until 29.01

28.01 Meeting with Suttung

- They think we should implement Bullet like they did Box2D with Cmake, and that should work.
- We have to implement our own physics interface for Bullet.
- Discussing how to implement the 3D, if we are to:
    - Creating our own separate rendering classes/architecture in a similar architectural way to how the current 2D rendering works.
    - Redesigning the existing classes and functions, making support for a third axis (XY+Z).
- They think we should first go for point one and then go for point two if we can.

- We might need to find a texture atlas, the one they use is a pay for version.

29.01 Meeting with Nowostawski

- For the project plan we should add in the result goals what we expect, so simple that our mums could understand it. High level wording, Avoid the tech aspects. A summary of sorts.
- Under the Effect goal for GUC, what kind of research?
- We need to discuss bullet. 3 sentences.
- In the report we should add introductions for everything we mention SDL2, Bullet etc.
- Also need to add in OpenCL if we are using that in the project plan, not just mention it.
- Under 3D Physics we can talk about Bullet
- Need to set a date for when we are to choose between the two points under **3D rendering.**
- Will we support 2.5D ? Mariusz suggest we don't, since our assignment is big as is.
- How will we deal with changes in the Engine from Suttung, changes might break our current work.

Meeting with Nowostawski 04.02

- Discussed splitting with the IS group, decided on splitting from us and check the other group working with Nox since they do Network.
- Wondered about what was happening with the mailing list
- Check how other engines has done rendering and representation.
- Nowostawski agreed that we should have simple scene up this week and a basic design skeleton. We should have limitations tight at first and then set wider as we work.
  How many assets can the asset-manager handle when we are doing timeline management, and such.
- The other group working with nox, didn't like the namespaces and loggers.

Meeting with Suttung 04.02

- Talked about the renderer in Nox.
  They suggested we made a separate renderer that handled the rendering for 3D. And then merged that into Nox.
- We might need our own scene graph since theirs are focused on 2D.
- Talked about the Directory structure and decided on that if we want to make changes to it, we suggest it to Magnus first so they can do it. This includes renaming files and such too.

Meeting with Nowostawski 11.02

- Discussed repo problems.
    - Instead of having Master branch as the “stable code” branch, we could use it as the development branch and have a second branch where we push the code after it’s been sucessfully tested.
- Scene graph
    - Talked about what the nodes should contain and how they should be used. We discussed how we had done it, and Mariusz said it was ok.
- Caching scene graph
- How to handle what is infront of camera:
    - Look where the position of camera is. From that you can calculate what is infront of camera and visible. Dot product, negative is behind and positive in front.
- Are we going to re-render things from scratch when it comes to time management?
- Need to remember to invalidate storaged stuff. And to do that we need to set a point for how long back you can travel in time.
- For static objects you can just cache position.
- For Dynamic objects we need to be clever and find a way.
- How will we integrate/Design physics? Two choices:
    - Point to the object in the physic engine from the scene graph
    - Or point from the scene graph to the object in the physics engine.
- We should have our three components up and running. They should only be basic.
    - Simple Renderer (done)
    - Physics (Main goal this sprint)
    - Scene graph (almost done)
- Create a simple demo when physics are done.
- Web page. Mariusz don’t think there is a deadline for it.
- TODO: invite Nowostawski to our repo.

Meeting with Suttung 13.02

- Use Namespace instead of Trd
- Pleased we added assimp as a submodule
- Remember to test on Linux and Mac too.
    - Remember to use / in all includes (Mac)
- Link
    - Physics to Transformation component to Render actor graphic, use bullet user data ptr(void ptr).
- They suggested we should use a separate repo for demos
- For the time manipulation they suggest we try first with:
    - Track all transformations
    - Then track changes in the matrices.

Meeting with Nowostawski 18.02

- We should start thinking about how to wire things. Since what we are creating is part of an engine. What to wire for just the demo and what to wire for people who want to use the engine.
- If the demo we create is complex, can we reduce that complexity?
    - We should try to make things as little complex as possible.
    - Create many small demos first and then put them together as a large one.
- We should start thinking about how we want to do the time manipulation.
    - Speeding up and slowing down time should be simple.
    - Reversing time or jumping back in time?
- When it comes to the thesis we should keep track of :
    - What do we add and why?
    - Why we added what we did and why didn't we add something?
        - Time constraints, difficulty etc.
- Need a bigger picture of the whole project now. Need to find out how far behind/in front we are. Use Gantt and find out why there are discrepancies.

Meeting with Suttung 18.02

- Talking about problems with our code they found.
- Sporaland will make a pull request soon of things he found that was easy to fix. These are changes that are needed for Mac to run properly and certain things that are good practice.
    - VAO
    - Shader
- Use renderData for managing VAO switching.
- Indenting when it comes to Namespace, needs to be fixed. Don't need to have the code so far out to the right.
- Should think about optimization when it comes to what's in front and behind the camera.
- GL_FragColor is deprecated and should be removed (check it after the pull request from Sporaland)
- Parse the tree then render so you don't render something you don't need to.
- Remember to draw things in the correct order to avoid unnecessary draw calls.
    - Draw things that are in front first then backwards
    - When drawing things with alpha you should draw that after the non-alpha mesh. Front to back.
- Look at deferred rendering
- Look at GlRenderTarget.
- Suttung suggested we merged in changes to Nox weekly.
- Problem with building assimp static since both nox and assimp uses Poly-2-Tri which creates two slightly different libs using the same name. Nox engine can't link properly because of this. Magnus tried to remove assimps Poly-2-Tri and link it directly to Noxs Poly-2-Tri but it didn't work because of the minor differences in the lib files. Suttung suggested we asked Mariusz about it.
- Something wrong with the rendering on Mac, the airplane is a plane (weird square).

Meeting with Nowostawski 25.02

- Might want to look at threading the Renderer.
- The other group working with Nox said they would look at doing some 3D? Mariusz will look more into this.
- Can delete a branch in bitbucket after merging into master.
- Might want to figure out if we are to use event queue system for collisions or create a function the game developers can use onCollision()
    - reasons?
    - load handling, what to do with different events under heavy load?
    - limitations of our system, test.
- Discussed threading. We should handle it. Not the developers. Might want to look into Future/Promise
- Discussed time manipulation.
    - Bullet is deterministic so should be able to resimulate an action with the same starting parameters.

Meeting with Suttung 25.02

- Pull request from Sporaland
    - Mac fixes.
    - Renderer fixes.
    - Still don't render properly on Mac, he will look more into it when he have time
- Pull request from Vik
    - Some fixes
- Look at activating openCL
- Discussed having the logic in it's own thread. Might destroy things for Suttung. But they have thought about it themselves.
- Discussed having the renderer in it's own thread. Might be possible but will it optimize much for us?
- They suggest we start with simple things for optimization.
    - look at running Bullet as multithread
- If we are going to multithread we might look at OMP or use c++11 threads.
- Regarding the Collision events, we should try both and see what is best.
    - event queue
    - overloaded function onCollision().
- Discussed memory problem with shared pointer. No conclusion, just gotta try finding it ourselves.
- Discussed texture. They suggested we start with one simple texture and then try packing them so we avoid binding textures all the time.

Meeting with Nowostawski 04.03

- Talked about Namespace/suffix. Nowostawski means we should use suffix. So that it's easier for any developers to use it.
  Strong side of open source is that people can change things in the engine.

- Thesis: Why we spent time on Nox?
  - time manipulation
  - why not time manip for Unreal?
    - Unreal wasn't open source at the time we started
    - Our employer was more interested in Nox, why they don't use Unreal, you will have to ask them about.
    - Big engine, will take time to learn.
    - we are focusing on tech not game.

- Talked a lot about api/interface for onCollision()
  Suggest we start looking at it from the developers side and see how we would like the collision from their side.

Meeting with Suttung 04.03

- Valgrind for checking uninitialized variables and for memory leaks.
- talked about why our demo was different on our laptops than on our stationaries.
  - windows 8 demands uniforms are uploaded correct and is used correctly.
- Try using gDEBugger
- How to create child actor from json file. No examples out yet, Magnus will send us an example later.
- Directory structure will be implemented like Magnus suggested. (maybe not before next week because of astral integration with nox).
- Talked about graphics and physics failure on our laptops. Might be because of uninitialized variables, this depends on the specific platform/environment..

Meeting with Nowostawski 11.03

- Nowostawski want to measure performance on game objects between Unreal, Unity and Nox. He will set up for Unreal and Unity. We will do it for Nox. Nowostawski will tell us when he has made it.
  - Number of calls
  - How long it takes total
  - code complexity.
  - Number of variables.
  - Number of if's and switches,
- Hypothesis is that ours is faster but more complex since we are at a lower level.

- How are we to create 500 objects? write them all in the Json file? Nowostawski suggest we ask Suttung if they have an editor or something. Can we create them programmatically.
- Maybe create an API for creating objects?
- Regarding the pseudo code for the collision:
    - Dont need to pass source since it has “this”. So destination and other parameters like force.
    - Lambda don’t need source either since it can be retrieved from reference
    - To have as few things to pass as possible since we don’t want to introduce bugs for the developer. Meaning we hide the wiring in the engine.
- Interface class for Actor to override the collision
- Check out Life is Strange game
- Work we identify during our work should be added as future work.

- What type of time travel do we want.
    - Just rewind backwards, stop and play normally from there on a new timeline-fork?
    - Rewind backwards, stop and play in parallel with the actions you did before rewinding?
    - Possible to go back in time, do something different, then step/rewind forward again and automatically get a different result?
- Look at different games, classify the different types of time travel.

Meeting with Suttung 11.03

- Actors can be created programmatically using actor definition(path to actor json file).
- Components can be created and added to actors using json objects. If the user don't want to load a json object from file or create one programmatically it is also possible to send in a nullptr instead of the json object setting the values manually.
- onCreate() function from Component are supposed to be called automatically after the actor is created which it is not doing.
- Suttung don’t have a Json editor, they want one but haven’t gotten around to it yet.
- isActive() in an actor says if it will be simulated by the world and if it’s visible or not. We should test this for us.
- Suttung suggest we should not use a interface class for collision but std function. They believe it was more flexible.
- You can have an interface but you would have that for a component not the actor.
- check gaffrongames.com (google fixed delta times)

Meeting with Nowostawski 18.03 - Nowostawski was not available this week.

Meeting with Suttung 18.03

- The View class is an interface towards the logic layer.
- A view can only have one controlled actor.
- Suttung haven't tested with multiple views in RenderSdlWindowView.
- Asked about how we should represent the different onCollision functions an actor can have. They suggested we should use a Map since there is need to find the correct onCollision to remove it. Speed shouldn't be an issue before you have a lot of them.
- There are four different control types:
    - Vector - a 2D vector with values between 0 and 1.
    - Strength - a linear value between 0 and 1.
    - Switch - a boolean value for on and off.
    - Toggle - sends a signal.

Meeting with Nowostawski 25.03

- Mariusz suggested we reprioritize our tasks relative to time manipulation. Deformable mesh isn't something that is very useful for this and is also really hard and should therefore have very low priority. We should reorganize and reassess for the next 3-4 weeks.
- We asked about threading the renderer since we are having problems with lagg when it comes to Animation and Mariusz said we should try to thread it and 4 actors with animation should be fine.
- We should make a decision on if we are making a game engine with a good API for game developers or making a game engine for game engine developers/tinkerers.
- Mariusz recommends we don't spend time on making a nice API but rather focus on the time manipulation module.
- We created a classification of different types of time travel.
    - We should also map those who aren't doable for the thesis.
    - Provide games or movies as examples for them.
    - Now we should split the big cases into small sub cases
    - We should then create sets of those who are covering the same cases and would need some of the same implementation.
    - Then we can choose the architecture that cover the most cases and what we would like to try with constraints and difficulty in mind.
- Nowostawski suggest we start cutting corners, as we are closing in on the time we should start with the thesis.

Meeting with Suttung 25.03

- Promised they would fix the Directory structure today
- Problems with Merging: Suttung added formatting class for logging. Internal crash in format.h. Haven't tested on Windows yet so might be that or it might be us. He will look into it.
- They are considering moving their repository over to gitlab.com. This wont have negative any impact on use.

- Controlled actors animation is regulated by a animation control component which receives movement event from the control mapper.
- How to use nox engine on eclipse in linux:
    - Build the cmake project using Eclipse CDT4 - Unix Makefiles option in the configuration settings.
    - Create a new Eclipse project and import in the Unix make files into the project.
    - Set Eclipse working directory to source folder.
- We discussed the threading over Renderer;
    - Maybe have the different processes on threads but that would require a substantial refactoring of the whole engine and would take much time.
- Nox builds both 64 and 32 bit
- Memory allocation:
    - Suggested we used std::Array to set of enough memory for time manipulation since it has pointers.
    - Vectors are also a possibility.

Meeting with Nowostawski 15.04

Nowostawski had a new “time manipulation class”. Person A gives time to person B, Person A loses the amount of time he gives to B.

We talked about the animation problem we had awhile back. Nowostawski mentioned how some let the physics take care of some of the animation. Say person A get shot in the stomach by person B. Where person A then either is animated to look right or that soft body is used to simulate that he was shot.

We should make a demo for models, physics, animation and time manipulation.

We should make a plan for the next three weeks from next wednesday.

Nowostawski suggest we prioritize in this order:

1. Finish time manipulation
2. Clean up the project.
3. Make demo
4. Find limits (How many actors we can run at once. How long time manipulation lasts and such. Remember to write down the specs and framework on the computer we do stress tests).

Appendix time spent on project.
Appendix everything you don’t know what to do with.

Make presentation a tech presentation not a design. But make the presentation pretty.

Meeting with Suttung 15.04

Let them know when and where the presentation is of the Bachelor assignment.

Cite before or after “.”
But always after.

Meeting with Nowostawski 22.04

- We should start writing about the most important stuff (time manipulation).
- We should plan so we iterate over the writing process. First time is a draft, next time look at spelling errors and restructure, same next time, and next time and so on.
- We can set references to other chapters we know we are going to write later, but not written yet.
- The security part is not very relevant for us. We could write about memory leaks and stuff, maybe.
- Deployment: for our context it’s not that interesting. However, we may discuss how our work affects the deployment of the NOX engine. Also discuss integration with NOX - will there be two separated NOX engines, one for 2D and one for 3D? How will the engine be maintained later? We should discuss this with Suttung.
- Move “Future work” part to the “Conclusion” section.
- Rename “Time manipulation manager” to “Time manipulation” under subsystems.
- Move section 4 to after section 6.
- Discussion: Personal thoughts.
  Part1: What we have done and what should have been done different.
  Part2: Reflection: tools, methodology(?)
- Conclusion: more concrete.

Meeting with Nowostawski 29.04

- We should use \n{note} for when we need to note things. Add a ‘d’ in front of ‘n’ to note that you finished it. \dn{note}
- We can repeat small things in the report or just refer to where it is written. Repeat can be used in things like chapter introductions.
- The design should be written for developer, while implementation should be written for people working with the engine. What is the toolbox? Concepts and facilities should be written. We should not use any class names here, except from the class diagram.
- Implementation should contain: language, algorithms and construction. What does an engineer need?
- Our diagrams need to be readable as a printed version. Either split or make it bigger/rotate it.
- Text in the diagrams should be same size and font as captions.
- Explain things we write about, we might understand it but the reader might not know.

- In discussion or testing phase we should have some figures or charts showing performance with different amount of actors.
- We speculate too much when writing. Fix this
- Write about the iterations we did with Suttung when it comes to testing. What did we have to change to make things work on Linux and Mac. This could be in Deployment under RUP or Testing chapter.
- RUP part: we do not need to refer to where in the thesis it is, that is overkill.
- Ideas that we tried, but failed on should not be in design or implementation. We can write about it in discussion.
- It's better to ask Mariusz to give feedback on chunks of text.
- In project management section, make clearer why we used Google Docs.?
- Use capital letter when writing the words "Chapter" and "Section". Always use this in front of the number/reference: "Section \ref{sec:troll}".
- Move process chapter before design chapter.

Meeting with Nowostawski 06.05

- Document structure should be last in the Introduction.
- We need to talk about the integration with Nox under Deployment
- Requirements needs to say what not how, so we should move the use case and the things for that to Design. Anything that is too detailed for requirements should be moved to Design.
- Abstract and Conclusion needs to be well written. This is probably what sensors will read first. Very important that this is written well.
- The conclusion has to give the essence of what we have done.
- Abstract should contain:
    - Topic/goal
    - What we have done
    - what results we have
- caption should be written like: \caption[short title]{description}
- Background could be written more academic, it is very "chatty" right now.
- Bullet points need to end with "." and start with capital letter OR start with small letter first and a "," after each point when listing things.
- The goal needs to be changed a bit; The target audience for the engine is … "type" developer. "Type" = core engine developer AND normal game developer.
- Avoid "()"
- Avoid things that refer to time. "At the point of writing" can be used.
    - "Currently" is bad
- Conclusion should contain the section about "time manipulation has little research".
- What kind of time we are focusing on.
- If we want Mariusz to read the thesis before the weekends we have to ping him before 5pm on friday.
- Avoid words like "more", it's not to the point and can't be measured.
- The thesis is riddled with past tense problems.
    - was and has been
- The main goal … of the inception…

- “But” is not a word we should start a sentence with.
- page 10. Area of time manipulation.
- Need to fix grammar problems, but focus on content first of all.

Meeting with Nowostawski 11.05

- Abstract:
    - Tell what Nox is and Noxplus.
    - Not how we worked.
    - More engaging.
    - Explain more physics and such.
    - Tell it like the reader don’t know what we have been doing
    - Start with second sentence.
- Remove the words “very” and “much”.
- Move time constraints under constraints
- Capitalize engine in “Nox Engine”
- Need to fix where we used the “GUC” macro
- Page 3. specify, loading from where?
- Keep terminology in main document
- Introduce something and then why. (ex: CMake)
- Specify if it is open source or not.
- Refer to the class in the doxygen. Can upload doxygen on website.
- Conflict solving ++ in the requirements
- Workload needs to be rewritten and added under 3.5
- 5.7 The tech stuff from time manipulation?
- Can talk about plan and execution in the same section.
- Future work: The bullet points should have some sentences with plan on how.
- Try and say more what the library did for us but put more emphasis on what we have done. Bullet/Assimp/Nox is ok.
- Highlight work we have done.
- Rewrite the part about the chaotic system. Problem is with the storage requirements.

Meeting with Suttung 12.05

- They wanted a pull request, from our repo to their to take in the new features.
- Should have split features into branches.
- Takes long time to go through everything, testing and reviewing it, before taking it in. So they do not have time to take it in at this time.
- Will test for Mac today.

Meeting with Nowostawski 13.05

- Abstract:
    - Noxplus and tasks needs a connection
    - Extension with what?

    - In this thesis we describe what Noxplus …
    - Solution 2:
    - 2 tasks, achieved it by extending nox -> noxplus
    - goals are not a good word, use features or functions. Since this is a software, use extension.
- Dont care about tech in the conclusion, so talk high level. Meaning the file extensions:
    - Maya, 3D studio files
- Careful with “implemented a physics engine”, use something like: “implemented a physics submodule/interface”.
- What does the demo have and demonstrates? “It is used for … and shows ...”
- What have we learned? Give some sort of evidence. Like lines of code from previous projects vs Noxplus. The corridor will do. Cite? Did we use new features?
- We should not talk about “having to cut things”. It weakens the thesis. SHows that we have now planned well. DO NOT ADMIT MISTAKES! We can however mention that we have learned to scope projects better for the next project.
- We have done a lot of work, but there are still work to be done. <--Future Work
- Explain combination of 2D and 3D.
- To tackle more actors -> exploit Bullet to it’s fullest, using OpenCL.
- Solving camera rotation using quaternions.
- Future work should be: problem -> solution
- Assimp lacks documentation -> evidence
- No “seemed”. Report facts.
- Deployment in implementation not discussion.
- Move Assets to implementation.
- “A bit more” rephrase 7.2.1
- All claims should have evidence.

I Credits

We would like to thanks:

- Ogldev, Etay Meiri for providing use with good tutorials on skinning animation and deferred shading.
- Opengameart, Danimal for the animated goblin 3D model distributed under the CC BY 3.0 license. No changes was made to the 3D model.
  License: (http://creativecommons.org/licenses/by/3.0/).
  Artwork: (http://opengameart.org/content/goblin-animated-by-motion-capture).
- AlpArt on TurboSquid.com for providing us with the free wooden box model. No changes were made.
  Licence: (http://support.turbosquid.com/entries/31030006-Royalty-Free-License?locale=1)
  Artwork: (http://www.turbosquid.com/3d-models/wooden-box-3ds-free/631645)
- Anton Gerdelan for good mouse picking tutorial.
  Tutorial: (http://antongerdelan.net/opengl/raycasting.html).
- Niven for mouse picking.
  (http://www.cplusplus.com/forum/general/135193/).
- Credits to Mike Shaffry and David Graham, from Game Coding Complete, fourth edition.

J Sprint review and retrospective meeting

Sprint 1:

Review:

- Sat up the project. Had some problems with Cmake but figured it out later.
- Camera is done. Have a bug with rotation. Low prio fix. Gimbal Lock. Can be fixed with constraining the angle or rotation with quaternion.
- Transformation component is done.
- Open asset loader(Assimp) is integrated into the engine, loading and displaying a static model.
- Created scenegraph for 3D.
- Created actor for 3D.

- The TrdOpenglRenderer class was not set as finished even though we had a simple scene up by the end of sprint since it wasn't as finished as we would have liked it to be as a basic renderer. It was the same deal with the shaders.
- The git repository was not marked as finish because the development branch had a few issues .

Retrospective:

- Bad at logging. Need to be better.
- Bad at using Jira. Need to be better.

Sprint 2:

Review:

- Prototype of bullet integration is done. (Init empty world and create sphere and plane collision shape).
- Problems with making Bullet a submodule for our project, turned out it was we missing some changes that was needed in some cmake files.
- Created components for Actors: physics and transformation change.
- Created component for Mesh.
- A prototype for bullet interface and simulation class was created but not marked as finished because the simulation class still needed to be restructured and refactored.

Retrospective:

- Bad at logging. Need to be better.
- Bad at using Jira. Need to be better.

Sprint 3:

Review:

- Implemented basic callback system for collisions between two actor, missing the event system.
- Put together a proposal for new folder structure.
- Implemented debug rendering for Bullet.
- Creating the BulletSimulation class was not set as finished. Most of the tasks is now completed:
    - Callback function
    - onSyncState function.
- Missing now only createShape function. It is created just missing the complex shapes. It has all the basic shapes.
- We haven't received the new directory structure from Suttung so the task to create namespace for 3D, commenting and refactoring the Rendering system was postponed.
- Rendering system was also postponed due to Even being on vacation and we would like to have him here when we do it.
- The demo was postponed because we wanted to create it when we cleaned up everything.

Retrospective:

- Bad at logging. Need to be better. (Seriously need to be better now)
- Bad at using Jira. Need to be better. (Seriously..)

Sprint 4:

Review:

- Implemented texture for 3D models. Currently loading new texture for each model, need to create a map to prevent the same texture to be loaded in multiple times.
- Refactored scene graph. Support for child nodes and each child will update their model matrix as the graph is traversed. .
- Introduced two rendering steps. The renderer will first draw all solid meshes, and finally it will draw the meshes with alpha textures (transparent ones). We did look at how to order the drawing of the alpha textures, but we need to work more on that.
- We didn't create new namespaces because we are still waiting for Magnus to create the new directory structure. We have also decided to not create new namespaces for 3D because the namespaces would make it harder to distinguish the 2D and 3D functions from the game developer's view. Using suffix on our classes would be a more tidy approach.

- We haven't commented every function yet because were waiting for the new directory structure form Magnus before will clean up the project.
- The demo was postponed again because we wanted to create it when we cleaned up everything.
- Implemented a callback system using the event system. Sending events throughout the system with broadcasting. Need to try different things.
- Still missing certain complex shapes in the createShape function. Haven't been worked much on.
- We didn't have time to start playing around with time manipulation.

Retrospective:
- We need to learn to close issues we are done with and add new ones when we see issues we need to address.
- Good at using Toggle!
- Need to get more sleep so we can work even harder.

Sprint 5:

Review:
- Cleaned up rotation, wasn't much to do.
- The physics/graphics issues on laptop fixed itself, we believe it was that we buffered things twice. Was fixed when deferred rendering was implemented.
- We wrote different Pseudocode for the API for onCollision so we could discuss the different ways to implement it and what way would be the best.
- We also wrote Pseudocode for the API for time manipulation. We are going to have to rewrite that one after discussion with Nowostawski.
- Almost done with deferred rendering. Handles easily many lights at the same time. Need to make a generic interface for creating lights, placing them and attaching them to other actors.

Retrospective:
- Getting better at closing issues, but we still have some way to go.
- Still good at using Toggle.
- Sleep handling still bad. But is also getting better.

Sprint 6 (7):

Review:

- There won't be added any more shapes before we finish the time manipulation module. Shape for deformable mesh was suggested that we dropped since it was really difficult.
- Need some feedback for the reuse of collision shape, we think we finished it but it will stay in the backlog until we gotten some feedback.
- The demo was postponed again because we wanted to create it when we cleaned up everything.
- We haven't commented every function yet because were waiting for the new directory structure form Magnus before will clean up the project.
- Still haven't had time for some of the items in the sprint: Cleaning up includes, commenting, triggers, optimizing deferred rendering, fix warnings, making lighting generic and checking the isActive function.

Retrospective:

- Getting better at closing and creating issues, but we still have some way to go.
- Still good at using Toggle.
- Sleep handling is bad again.
- Need to prioritize tasks and plan a few sprints ahead.
- We didn't manage to get much done in sprint 6 so we decided to extend it into sprint 7. This was mostly because the task we worked on was too large. So we need to be better at splitting the tasks up into smaller ones to see how much work a task really is. We also need to increase our working hours, we have been a little lazy. We believe this is due to that we don't see much progress. The demos would have helped showcase this but since we don't have the directory structure yet we been postponing it.

Sprint 8:

Review:

- OnCreate() was fixed, we forgot to test if an actor had a Transform3d instead of transform component.This also fixed the memory leak.
- Reuse of collision shapes was said to be ok.
- Demo was postponed again.
- We cleaned up some code and commented some. Not done yet.

- Triggers and kinematic shapes are down prioritized.
- Started mapping time manipulation. Decided what types of time manipulation we wanted to support.
- Time manipulation module was done in sprint 9-10. Not in 8 as Jira says.

Retrospective:

- We completely forgot about Easter, we should have taken this into account when planning the sprint.
- Håkon left for The Gathering and there was no meeting with supervisor/Suttung on the day we were supposed to close the sprint, so we didn't set up a sprint for the last part of the Easter. We were still working, but we should have closed and set up a new sprint.

Sprint 9-10:

Review:

- Created a component for lighting.
- Created classes for logging and managing time.
- Static rewind and forward playback is done.
- Sat up LaTeX, so we can start writing.
- Pausing the game/logic works now.

Retrospective:

- We should branch more often when developing new engine features.
- We are still too lazy with Jira.
- Need to reprioritize again, and plan for the last month.
- Given up on working from 8-16, we are now working from 13 to after midnight.

Sprint 11:

Review:

- Cleaned up and commented code.
- Created the Demo
- Refactored world logger to use List instead of array.
- Implemented conflict detection.
- Sat up thesis structure.
- Added support for spawning new actors during gameplay.

Retrospective:

- We started on the thesis so we extended this sprint by 2 extra weeks.
- Increased our working hours to over 50 hours a week per person.
- We should have branched for each feature we created to help lessen the workload for Suttung.

- We are great at using Toggl.
- Still terrible with closing tasks that were done.

K Logged Hours

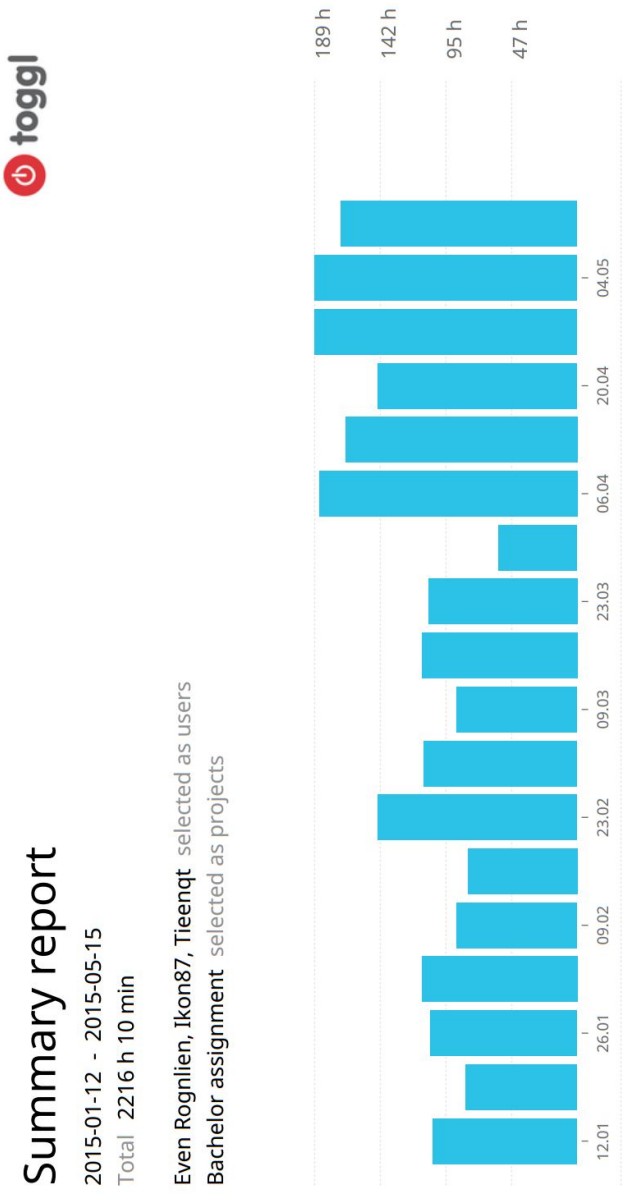


Figure 41: Summary report from toogl.

L Nox control system

```
/**
 * Generates logic::control::Action events based on the keyboard input.
 *
 * The mapping from keys to events is loaded from a JSON file with loadKeyboardLayout().
 * There are four types on control actions (all based on logic::control::Action, so see there for more info):
 * - Vector.
 * - Strength.
 * - Switch.
 * - Toggle.
 *
 * ## Key Strings
 * For mapping a key string to an actual SDL_Scancode, the SDL_GetScancodeFromName() is used.
 * So all key strings must match a string in SDL_GetScancodeFromName(). See http://wiki.libsdl.org/SDL_Scancode for all
 * the possible values.
 *
 * Since SDL_Scancode is used, the keys are independent of the keyboard layout. So mapping WASD to actions would result
 * in the same physical location on both a qwerty and dvorak keyboard. The qwerty layout is always the layout used for SDL_Scancode.
 *
 * ## Control Types
 *
 * Each control type has its own property in the JSON file:
 * - Vector: __vectorControls__
 * - Strength: __strengthControls__
 * - Switch: __switchControls__
 * - Toggle: __toggleControls__
 *
 * ### Vector Controls
 * The vector controls is a set of actions and directions belonging to these actions. For example there can be
 * an action "move" and four directions vec2(1, 0), vec2(0, 1), vec2(-1, 0) and vec2(0, -1) (there is no limit).
 * All directions have a key mapped to it. For example for "move" you could have D, W, A, S matching the directions
 * mentioned above. Combining several directions will average the vectors. For example holding W and D will trigger
 * a direction of vec2(0.7, 0.7). All directions are normalized.
 *
 * The format for the vector controls is as follows:
 * - __actions__:array[string] - A list of all the available actions. An action used in __buttons__ must be listed here.
 * - __buttons__:map[array[object]] - Keys mapping to arrays of direction actions.
 *     + __action__:string - Name of the action that this key will trigger.
 *     + __vector__:vec2 - The direction of the action that this key will trigger.
 *
 * ### Strength Controls
 * These are mapped as switch controls where pressing a key will result in strength 1, while releasing will result in strength 0.
 * See Switch Controls for more.
 *
 * ### Switch Controls
 * A switch control is either on or off. It is off when a key isn't pressed and on when a key is pressed. The JSON format is as follows:
 * - __actions__:array[object] - All the actions available.
 *     + __name__:string - Name of the action that will be triggered.
 *     + __keys__:array[string] - All the keys that will trigger this action.
 *
 * ### Toggle Controls
 * A toggle control has no state, it is just a plain signal. Only a key press will trigger a toggle action. The JSON format
 * is exactly the same as Switch Controls.
 */
```