Flying Samurai – Programming documentation

Jan Beneš, Oskár Elek, Marek Hanes, Ján Zahornadský

June 3, 2010

"We make war that we may live in peace." — Aristotle

Contents

1 Introduction

This is the non-user documentation of the Flying Samurai project, which has been developed at the Faculty of Mathematics and Physics of the Charles University, Prague in order to fulfil the requirements of the NPRG023 - Software project course.

We put a lot of effort in making the document what we ourselves would want it to be. We hope the level of detail is high enough to provide the crucial information that would be hard to gather without any kind of documentation, yet low enough not to flood the user with specifics that would make this document way too bloated and hard to maintain – that's what source code is for.

The document is divided into sections. After this introduction chapter, there is a chapter giving an overview of the project's whereabouts. The interested programmer should definitely take a look at the second and third chapters; the second chapter will tell you all you need to know to about setting up an environment for the project and getting it to build; the third chapter then contains the programming documentation, where an overview of the project as a whole, as well as of all the subsystem, can be found. In the next two chapters, our thoughts on the project as a whole have been summed up and people who have helped us with the project are mentioned. After that, the references are given. The document ends with several appendices containing auxiliary information and documentation of the project's tools.

2 About the project

This section will try to convey the whereabouts of our project. Special focus will be on the history and on the key decisions we had to make.

2.1 Team

The project started with five people – Jan Beneš, Filip Bureš, Oskár Elek, Marek Hanes, and Ján Zahornadský. Before the project was officially started, but after the specification was handed in, we have, by mutual consent, terminated our cooperation with Filip Bureš. Despite our efforts, we were unable to find anybody else with the required skills and decided to finish the project in just four people.

Initially, the work was divided as follows:

Jan Beneš Team leadership, game logic programming, misc. programming.

Filip Bureš Physics programming.

Oskár Elek Team leadership, graphics programming.

Marek Hanes Network programming, menu programming, misc. programming.

Ján Zahornadský AI programming.

In the end, the responsibilities were distributed a little bit differently, namely:

Jan Beneš Team leadership, externist (we use the probably incorrect recruitment, researchers and beta-tester management, game logic programming, menu programming

Oskár Elek Team leadership, artist management, graphics programming.

Marek Hanes core programming, menu programming, bug hunting, configuration management.

Ján Zahornadský AI programming, physics programming

The workload was rather evenly distributed amongst all team members and in the long run, it cannot be said that anybody leeched off the work of others.

We imposed a weak hierarchy on the team. The team leaders' role was mainly to keep track of where the project is, what needs to be done, and what needs to be fixed. Also, since the rather large amount of externists, the team leaders split the management responsibilities based on the areas their worked on in the game. In the end, everybody found a role in the team that he is good at.

We met every week once the design document was ready. The meetings usually lasted between 1.5 and 2.5 hours and kept everybody up to date with the project's progress. The fact that those meetings could be held in Faculty's projector equipped Lab Room was a big plus and made our work a lot easier. We also tried to establish a mailing list and a forum. The mailing list was meant to inform people, especially the externists, about the project's progress, but never caught on and was abandoned after about two months. Later, one of the externists got us a hosting for an Internet forum; it too got abandoned after a few months.

Finally, we list all the people who have shown interest in working on the project and the amount of work they did, and the reason they quit. We only list the people who never contributed to the project by their

initials. We hope that this will serve as a small case study for anybody who is willing to take the same path we did.

Also, we mustn't fail to mention our supervisor, Mr. Otakar Nieder, who patiently followed our progress, attended our weekly meetings throughout the duration of the project and always supported us. Thank you!

2.2 Externists

This short section describes our experience with the people we were working with. One might find it interesting, especially if going in the same direction as we did and plans to manage people working on a bigger project.

The strength of the World War I. airplane simulator community was one of the reasons we decided to do the Flying Samurai instead of other project we had in mind. A quick research showed that a large number of people still play and modify rather arcane games such as Red Baron II and Red Baron 3D [25]. Also, the only reasonable alternative we could find was Over Flanders Fields [21], a mod of Microsoft Combat Flight Simulator 3 that has already been in development for a rather long time, needed the original Microsoft software, and seemed to be stalling at the time we started the project.

We have made short announcements and advertisements at The Aerodrome (`http://www.theaerodrome.com`) and at Wings of Honour (`http://www.wingsofhonour.com`), hoping to attract researchers, beta-testers, flight-model testers, artists (also advertised at the `http://blender3d.cz` and other Blender oriented websites), and menu artists (for that position, we advertised at `http://cegui.org.uk`). Later, we even tried mailing individual people, especially artists, with request for help. Our policy was that you are free to leave whenever you want as long as you let us know that you are leaving.

Below is an attempt at a complete list of external people who cooperated on the project in no particular order. For each of them, a short history and work overview is given. Names of people who are not in the credits have been anonymised. Two of them actually left for medical instead of personal reasons, but we decided not to implicate those people directly.

Manilo Japson A Belgian modeller and the externist who contributed the most to the project. Only got in touch late in the project, but got a lot of quality work done. He did two complete airplanes models and all of the buildings.

Matt Haslem A Canadian modeller who did the SE5a model. After finishing this model, he complained about personal reasons and never replied to our mails again.

Marek Moravec A Czech modeller whom we recruited from the Czech Blender site. Did the Fokker E III models, without the unwrap, and then quit due to a change in priorities.

Darren Reynolds A 2D artists whom we contacted directly. He did a lot of research in the few months he was with us, the complained about personal problems, and then quit responding altogether.

J A Swedish modeller. Did some work on one model but quit for personal reasons and we never got our hand on any of his work. Stayed in touch, but due to a change in priorities never came back once his problems got resolved.

G A Czech modeller whom we recruited from the Czech Blender site. Did a model (of which we saw some screenshots) that was lost in a hard-disk crash before he managed to send it to us. Didn't contribute to the project again due to workload and a change in priorities.

K A Czech modeller whom we contacted directly. Wanted to cooperate once his current project is over, but changed his mind about a month later.

L A beginner artist who has shown interest in working on the project, but never replied to one of our mails about 4 days later.

Sebastian Dreyer A history student from Germany who volunteered as a result of one of our advertisements. Helped us with the historical research early in the project, stayed for three months, and left because the actual workload for the project was higher than he expected.

A A high school student from Great Britain, wanted to help us with testing and research. Got in touch early in the project on V's recommendation. Stopped replying when first real work was needed about 9 months after he first wrote to us.

V A high school student from the Czech Republic who wanted to help us with testing an research. Got in touch early in the project. Also tried modelling, but left due to a change in priorities about 10 months later, without every contributing to the project.

P An American who wanted to do research. Complained about workload, then stopped replying altogether. Never did any actual work, left 4 months after getting in touch.

Mikuláš Peksa A friend of one of the programmers and a physics student at MFF UK, advised us on the physical model throughout the project.

Tomas Caitthaml A friend of one of the programmers and a former student at MFF UK, he dedicated a day to on-site testing a few days before the project's hand-in.

János Sebök A tester from Hungary who got in touch rather late in the project. Due to technical difficulties on our side, beta-testing couldn't start as early as we had hoped. Nevertheless, in the 2 months he stayed with us, he did a lot of valuable work.

F An Australian high school student who wanted to do some modelling. First postponed his work due to school exams, later due to personal reasons, and finally left due to a change in priorities.

S An English speaking artist who wanted to contribute some already existing models on the condition we optimize the game to run on his 32 MB graphics card. We never got our reply to our rather long explanation of why that is not possible.

C A Columbian music composer who volunteered to do some music. Stopped replying about 14 days after most details about the music were agreed.

D A German volunteer who even wanted to visit us during his trip to Prague. Didn't get in touch as agreed on the expected meeting day, quit a few days later due to a change in priorities.

J A friend of one of the team members. Did some work on a 3D models, but he didn't show any further progress after the model was about half-way done.

R A texture artist from Russia. Left us after about 14 days after starting due to personal reasons.

Scott Hamill The webmaster of The Aerodrome (`http://www.theaerodrome.com`) who helped us with project publicity and let us use some of The Aerodrome's textual resources.

Ruediger Gemmrig The webmaster of Wings of Honour (`http://www.wingsofhonour.com`) who helped us with project publicity.

As you can see, despite our requests to inform us about their intentions to leave the project, the majority of the externists never did. Overall, most people do not have an idea of what it is like to work on a project of this size. They expect instant results, have very little sense of responsibility, and are not able to estimate the expected workload well. The crucial point probably comes with the first serious work request, i.e. during

the transition from being part of something to making sacrifices. Also, beware of people who are keen to list their preferred features, but refuse to put in any serious work or only work on things that never got requested but that they find interesting.

The externists, though, might have also had their reasons to leave the project. We got behind advertised schedule several times and late in the project, we had problems with running the game outside of the IDE which delayed beta-testing by several weeks.

2.3 Review of specification

Here, we will review the features as they were given in the specification [7].

- **Network mode with multiplayer support** - *"focus on LAN, we'll need to implement the server, performance optimization"*

  There is **no** network mode in the game. We had to drop it about 5 months before the project was due as it became obvious that there is not enough time left for it to get implemented.

- **Graphics and graphical effects** - *"we'll need to build the game over the Ogre3D engine, implement all the necessary shaders, program additional effects (tunnel vision, bloom, etc.)"*

  We built the game on top of the Ogre3D engine, implementing all the necessary logic, shading, and the two above mentioned effects. Refer to the programming documentation, Section 4.5.

- **Terrain with cities** - *"large scale terrain, we'll need to procedurally generate simple cities to make the game environment look realistic"*

  We use real-world height-maps, but the whole terrain is covered with procedurally generated cities, roads, forests and fields (Sections 4.5.3 and 4.5.4).

- **Realistic airplane physics** - *"lift, other basic effects, integration of the Bullet engine, that (like other similar engines) doesn't support flight physics calculations, need to ensure numerical stability of the calculations"*

  The game uses a rather realistic physical model that is capable of simulating most of the basic effects. Bullet engine is used for collision detection and to simplify physical calculations (see Section 4.6). The physical model shows no signs of numerical instabilities.

- **AI** - *"basic usage of tactics, need to do AI calculations in 3D"*

  The AI groups the airplane in formations, makes them follow 3D paths and is able to engage in aerial combat in 3D (see Section 4.7).

- **Different mission types** - *"ace dogfight, squadron dogfight, bombardier raid, bombardier escort, patrol the front"*

  All of the above mentioned missions types have been implemented (see Section 4.4.2).

- **Career mode with campaign** - *"front positions change with time, military ranks and transfers to different airfields, basic statistics"*

  All parts of the career mode where changing the parameters with time makes reasonable sense have this ability (see Section B for more details). Military ranks, as well as awards, are implemented on a per nation basis (see Section B). Transfers, both requested by the user and enforced by the game logic, have been implemented (see Section 4.4.3). The game keeps track of a wide range of the player's statistics.

All in all, with the exception of the network mode, we have reached, and sometimes exceeded, the given specifications.

The network mode was added to the specification to provide enough work for all of the team members. With the departure of Filip Bureš, the responsibilities within the team changed (see Section 2.1) and even though we have made every effort to include the network mode in the game, it had to be dropped so that we could concentrate on the more important parts of the game.

2.4 Hardware requirements

In agreement with what has been announced in the project specification, the hardware requirements of our game are quite high. They are still lower that requirements of contemporary 3D games, but of course, we don't reach the visual quality of these games either. There are three reasons for this. First, we use the Ogre3D graphics engine, which is a general 3D engine not optimized for specific purposes. All modern game engines are specifically optimized not only for real-time 3D graphics in games, but also for a concrete type of the game (shooter, strategy game, simulator etc.). Naturally, we don't have access to this kind of technology. Second, the game contains a really large terrain, larger than the great majority of even commercial games. To render this terrain, we use a terrain rendering plugin for Ogre3D, which is not designed for such large terrains, yet is the only usable mechanism to render paged terrains on Ogre3D. Third, we simply don't have time and resources to optimize the game beyond configuration of the third party engines and libraries we use.

The following hardware and software configuration is the recommended minimum to play the game without any significant degradation in speed and quality.

- **CPU:** At least 2GHz, preferably multi-core CPU. The game can utilize up to four cores.
- **RAM:** At least 1.5GB on Windows XP, or 2GB on Windows Vista/7. The most of the memory consumption is caused by the terrain, of course.
- **VGA:** NVidia GeForce 8600GT or equivalent with Shader Model 3.0 support and 256MB VRAM.
- **HDD:** Around 2GB of HDD space for the installation.
- **Controller:** Standart keyboard and mouse. We also support joystick (without advanced features like force feedback or PoV hat).
- **Software:** Windows XP/Vista/7 (with support for x86 applications), DirectX 9.0c runtime (February 2010 or newer, because Ogre3D 1.7 needs it), .NET runtime (for the Startup configuration dialogue).

The main bottlenecks of the game are graphics and physics. Graphics is the most prominent consumer of resources, but on configurations with strong VGA and slower CPU, it can be the CPU which bounds the game performance (one example is the machine of one of the developers – 1.86GHz Intel Core2 Duo CPU and NVidia GeForce 8800GT VGA). The graphics part is most resource-intensive when a lot of terrain geometry (such as mountains) and particle effects is on the screen at the time, while physics start to be demanding when there are many airplanes in the mission at once, especially when some are in contact with the ground (because of the collision calculations).

2.5 Comparison with similar software

One of the reasons we chose to work on a combat flight simulator from the World War I. was that while there's a great abundance of shooters, strategy and RPG games on the game market, there is not even

remotely as many flight simulators, and even among these, flight simulators from the First World War are even sparser (indeed the number of people keen to play flight simulators is lower as well, but still). However, some similarly aimed project exist, and a few of them we summarize here.

Red Baron II, Red Baron 3D Red Baron II first saw the light of day in 1997 and a heavily patched version, Red Baron 3D, got published in 1998. One of the latest patches significantly improved multi-player capabilities of the game, bringing the number of players who can play the game at one moment to 76. This led to a number of virtual squadrons being established and the game was, due to it's easy modifiability, kept up by the community until at least 2008, for about a decade. Other than that, the game also included a dynamic campaign mode that allowed for hours of single player game-play. In 2009/2010, Mad Otter Games purchased the rights to Red Baron 3D, and efforts to consolidate all the community patches are underway [24].

The first installment of the Red Baron series was our main inspiration. We obviously surpass Red Baron 3D's limited graphics, offer at least the same amount of modifiability, and are not far from reaching it's career/campaign mode, but we provide no networking capabilities.

Rise of Flight [26] A relatively new (Q2 2009) WWI combat flight simulator from the Russian team *neoqb*. It is currently considered to be top-of-the-line WWI flight simulator, with very convincing graphics and realistic flight model. We can't compete with this game in the quality of these aspects, but we are not that remotely behind it either (e.g. we use similar techniques, such as terrain splatting, in our game), and we are of course speaking about a commercial product here. Moreover, RoF had been in the development for 4 years and the team has about 30 developers [31], and even now, almost a year after its release, the development team is releasing large (hundreds of MB) patches and updates for the game, to remedy bugs within the game and enhance the amount of features to make it more interesting. The game is available for roughly 40 EUR.

Over Flanders Fields [21] OFF is a commercial third-party modification of the Microsoft Combat Flight Simulator 3 game by *OBD Software*. Unfortunately, we haven't been able to try it, because we don't have an access to MCFS3 and we are not aware of any demo version of OFF either. Judging from the videos, screenshots and reviews, the game is similar to Rise of Flight, and therefore stands in a similar relation to Flying Samurai. As for the development, the game is still being developed, although some earlier retail versions have already been released – and as we said, this is just a mod of MCFS3, which means that the core features of the game engine didn't have to be written. The price is 30 USD plus the price of MCFS3.

2.6 Timeline

Specification We decided early in the winter semester of 2008/2009 that we are going to make a computer game as our Software project. For that reasons, we all signed up for the Computer Games Development course, led by Mr. Otakar Nieder. We dedicated the whole semester to writing the design document, successfully defending it at the end of the course.

Early project stage Preparations for the actual implementation, such as middleware research, and the architecture of the project, have been done during the summer semester of 2008/2009. At the end of the semester, after only a few unanswered questions were left, we have submitted the specification to the Project Committee on 6. March 2009.

Summer of 2009 Shortly before the end of the summer semester of 2008/2009, on 13. July, we made a formal request to start the project. By that time, we have already parted ways with Filip Bureš and were rather certain that we wouldn't be able to find a suitable replacement.

While we were hoping to make a lot of progress over the Summer of 2009, unexpected personal reasons and the Summer itself kept our work rate below our expectations. At the beginning of summer, we were hoping to have the project ready before the end of the year. At the end of the Summer, that wasn't the case any more.

Winter semester of 2009/2010 The winter semester of 2009/2010 was the time when most of the hard work was done. We built on the core that had been written over the summer and integrated most of the other libraries with the game. By the end of the year, we had an alpha version of the game ready. It was at that time also that we had decided to drop the network mode, as it became obvious that there was not enough time left for that.

Summer semester of 2010/2011 During the summer semester of 2010/2011, the game's functionality was finished, graphical resources integrated, datafiles created and integrated, and documentation was written. During this time, our progress was hindered for about 14 days by the crash of the `urtax` server, which hosted our SVN.

Luckily, we didn't qualify for the first hand-in date by a few days, which provided us with some additional time.

Summary From the time we first met until the hand in date, about 21 months have passed. That amounts to about 17 months of work on the project, from start to finish when the idle months (exam periods and similar) are not counted.

In comparison with the specification, which state 2. January 2010 as the expected end of project date, we have needed extra 5 months. Yet, we have still managed to fit within the given boundaries.

2.7 Future of the project

We will, most probably, release the game as open source shortly after it is defended. From the initial feedback we got from our testers and friends, it seems that the game has enough potential to convince some people to add further functionality.

"...its an enjoyable game really (at least for a simmer). I spent 2 hours fighting this morning ;). Its quite an accomplishment you guys have made there. Physics is reasonably good (with the observations I made earlier); I have some background on the complexity of flight physics calculations, and I am aware of the compromise you need to make between realism and performance. The graphics is also ok; of course you can not compete with commercial high budget projects but you created some good-looking terrain. Also in a flight sim I have always appreciated the physics more than the effects."

— János Sebök

Also, the game has been developed with easy modifiability in mind; adding new airplanes or creating new campaigns should therefore be rather easy for a skilled user.

2.8 Known bugs

Despite we made all the effort to make the game as stable as possible, there may be some problems we haven't remedied, mainly because of the strictly imposed deadline, and also because of difficult invocability or rareness of some of them. Also, there are some glitches we are aware of, but are very problematic to solve (for instance because they are caused by 3rd party libraries or some other external factors). We list the ones we know about here, which of course means that they will get a highest priority of repairing when we issue a patch. Feel free to report anything not listed here to one of the e-mail contacts listed at the project website `http://tinyurl.com/flyingsamurai` (and please attach the `Ogre.log` file from the `config/` directory and a detailed description of the problem you've encountered).

- We've had some issues with ATI graphics adapters. Officially, Ogre3D supports them, but there is a lot of factors which may influence graphical issues (drivers, current version of related libraries, hardware errors etc.). To be more specific, we have problems with airplane mesh displaying on one concrete piece of ATI HD3200 Mobile card. Unfortunately we can't test this further, because we don't have an access to machines with ATI cards.
- On one or two particular machines (in the MS laboratory), we've encountered problems with Direct3D device losing (causing the game to crash), in particular when performing Alt+Tab, but also during a normal flight. This was happening very rarely, and we suspect driver problems.
- We started to work on our sound subsystem just recently, which implies that it is still being in an experimental phase. The machine gun sound is a bit noised sometimes, and in a single occasion we've experienced a crash of the game, when an enemy airplane crashed into us from behind (which is in itself very rare event, as our AI is a good pilot overall). If some problems with the sound occur, we recommend to disable it from the Startup dialogue.
- When transitioning from one terrain page to another, the game lags slightly. This is due to the loading of the new terrain data for the next page. We can't easily influence this, because it is an internal matter of the Myrddin Landscape Plugin (which claims to support background data loading, but apparently, it doesn't work).
- The freelook camera mode (controlled by right mouse button) sometimes gets stuck and it's necessary to hit the button once more to dock the camera back behind the player's airplane. This is most probably caused by skipping one game frame in the graphical subsystem and with it, also the information about the event that the right mouse button has been released (this may happen in lower framerates, when the graphics is running slower than the game logic).
- The Myrddin Landscape Plugin doesn't support per-renderable changing of the camera clipping planes (which influence numerical precision of the depth buffer). We therefore must use fixed positions of these virtual planes, which causes artefacts like Z-fighting of distant ground objects and the possibility to "peek" underneath the terrain, when the camera is close to it.
- It is possible to break the integrity of the game logic by filling nonsensical values into the configuration files, namely `Entities.xml` and `Media/Objects.xml`. We do some integrity checking of these during the game initialization, but we can't provide a full resistance to errors within these, so we recommend to stick with our directions with regard to the adding of new records into these.
- The game consumes quite a lot of memory. On some machines with low amount of physical memory, the game may crash on a `bad_alloc` exception, since no checking of available memory is done in most of the code (this is mainly an issue of the Myrddin Landscape Plugin, which consumes a lot of memory due to the fact that it handles the rendering of our large terrain).
- The SE5a airplane is untextured. We just couldn't find anybody, who would do an UV unwrap and draw those textures (doing UV unwrap is not trivial and we don't have an experience for it).

- The houses in the cities are misaligned with the streets sometimes. This is caused by wrong orientations of the building models we have. We didn't manage to fix this, because the exterist who was working on the house models delivered them just before the project deadline. We have included an ad-hoc rotation offset for the houses directly into the code to remedy the situation a bit, but the problem is still apparent.
- When using the Startup dialogue and running the game in window, the game window doesn't get activated and needs to be selected manually. We have tried to solve this problem by using several approaches, both from the Startup application and from the game itself, yet none of them worked.

2.9 Some statistics

In this section, we provide some statistics and estimates. First, it should serve as a reference for others who will be working on similar software projects; second, it is a small memorial to the work we have done.

As mentioned in Section 2.6, the project took 21 months from the first idea to hand-in, about 17 months of work. We have held approximately 50–60 team meetings, wrote about 68 thousand lines and just under 1.3 MB of C++ and C# code. Taking into account just the development period – some 11 months – and a workload of 15–20 hours per week and person, we get to 660-880 hours per person. We wouldn't be too surprised if the real workload got to somewhere between 1000 and 1500 hours per person for the entire project. We have made about 880 commits on the `urtax` server, 265 commits on our new server, and over 185 commits on the documentation repository.

3 Building the project

3.1 Setting up the environment

First, you'll need to download the source-code from the SVN repository. The current address of the SVN repository can be found on our webpage (once the project has been defended): `http://tinyurl.com/flyingsamurai`. Alternatively, you will already have gotten all the necessary files via official channels.

You will also need the following tools to successfully build the game and/or game tools:

- CMake 2.6 or higher [4] - a build system that provides customized makefiles/project files for a variety of environments.
- Microsoft DirectX SDK 9.0c [5], at least February 2010 (**!**) - SDK required to successfully build and run the game.
- Microsoft VC++ 2008 SP1 Compiler [29] - compiler used for both the development and the release version.
- Microsoft Windows SDK 6.0 [30] - if you use Express versions of MSVC, you also need this.

Once you have all the tools installed and the source code with pre-build external binaries checked out of the SVN repository, change your directory to `\flying-samurai\config` and execute `build-game-vc9.cmd`. After the build finishes, you can find the binaries in `\flying-samurai\bin\Debug` and `\flying-samurai\bin\Release`. To run the game, the `PATH` environment variable must contain references to the 3rd party libraries that can be found in `\3rdparty*` directories.

One way to run the game is using the Microsoft Visual Studio solution file generated by CMake. For development, we use `\flying-samurai\FlyingSamurai.cmd`. The script adds 3rd party directories to `PATH` and allows Visual Studio to successfully launch the executables.

3.2 Building from sources

The game uses CEGUI, Ogre3D, OIS, Boost, Bullet, tinyXml, irrKlang, Myrddin Landscape plugin and PagedGeometry plugin. OIS, CEGUI, and irrKlang are used in a binary release form. Boost, Bullet and tinyXml are built unpatched. Ogre3D, Myrddin, and PagedGeometry are patched against known bugs and functional requirements. We provide pre-patched source code of Ogre3D and its plugins.

After checkout of the complete repository with 3rd party source codes, you need to execute the following steps:

- Ogre3D - Build with double precision and full threading support using Boost. Use the out-of-source build to the `OgreBuild` directory.
- Myrddin plugin - found in the Ogre3D plugins directory.
- PagedGeometry - found in a subdirectory of the Ogre3D directory.

CMake scripts are configured to distinguish the pre-built configuration from the custom build configuration. To open the project, you just need to execute `FlyingSamuraiEx.cmd`. Header files, library files and binary files are automatically used from your custom-built Ogre3D.

Our patches of Ogre3D and its plugins address following issues:

- **Ability to track every memory allocation using Ogre3D's integrated memory tracker** - we accomplished this by modifying the allocator interface and then refactoring the whole Ogre3D library and all of its plugins. Before, the majority of all allocations was done anonymously.
- **Resource allocation and deallocation logging** - for easy identification of leaking materials and textures.
- **Paged geometry optimization** - there are only two types of objects and they had thousands of instances in the game. This was the plugin's expected behaviour, but it had to be fixed nevertheless.
- **Incorrect usage of Ogre3D allocators** - both Myrddin and PagedGeometry contained incorrect calls to Ogre3D memory management routines resulting in the inability to correctly free game resources and quit the executable.

Temporarily, we also modified the Bullet library allocator. This, however, was very late in the project development and was used to find physics related memory leaks. These changes were removed since it would leave several places of non-game code non-compilable.

4 Programming documentation

The text below should give the interested read a general overview of the whole system and give him a short walk-through of each of the subsystems. Nevertheless, this is not a complete in-depth guide. The reader is encouraged to look at the source code and use, for example, the "Find all references" and "Find in file(s)" features of the IDE. We have used this method with a lot of success when the need to fix or integrate older code arose.

4.1 Architecture

Currently, the game is separate into these subsystems:

- **Entities** - a read-only database of game data.
- **Input** - input sampling.
- **Graphics module** - rendering.
- **Physics module** - physical computations.
- **AI** - AI computations.
- **Game logic module** - game state, career, menu, and similar.
- **Shared data** - a collection of data that the other subsystems share.

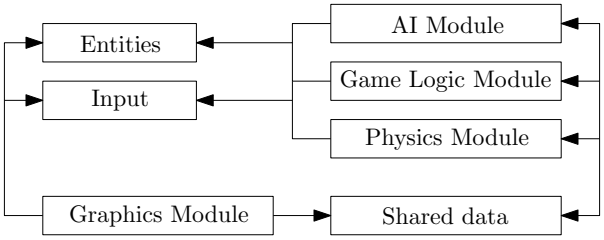


Figure 1: Dataflow between the models. The arrows depict the calls that request data, in other words, they go against the data flow.

When the game is running, the modules "exchange" information (see 1) mainly via the Shared data subsystem (see Section 4.2.5). The Entities subsystem is always read-only, while the Input subsystem is usually read-only.

In addition to that, communication via messages (see Section 4.2.7) is also possible and rarely, other forms of communication have been used.

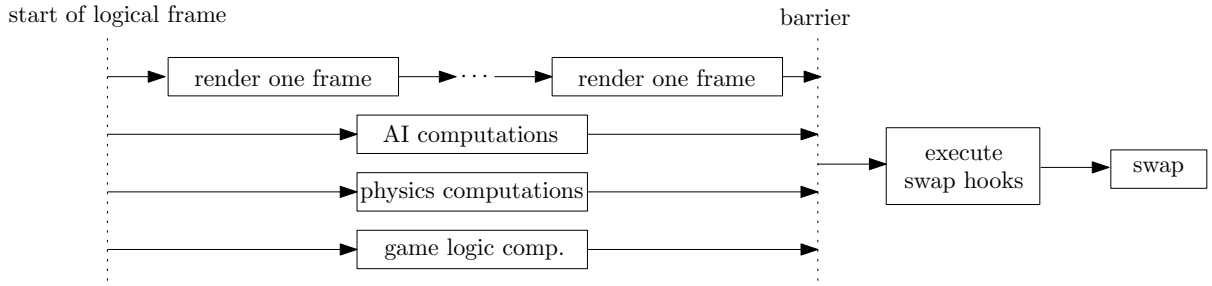


Figure 2: Control flow in a logical frame. All threads are waked, each of them does some calculations and stops at the a barrier. One of the threads (possibly a different one every logical frame) then executes the swap hooks and the swap itself. Notice that several graphical frames might get executed during a logical frame.

Since the game was designed to be multi-threaded, the architecture strongly reflects that. The modules correspond to individual threads (see Section 4.2.1), while the Shared data subsystem provides a mean of sharing data and the remaining two subsystem, Input and Entites, provide read-only data. At the end of each frame, swap hooks – functionality that is best implemented in a single-threaded environment – get executed (see Section 4.2.6) and the swap operation takes place (see Sections 4.2.6 and 4.2.4). The computation is done in cycles, called (logical, as opposed to graphical) frames. The control flow within a logical frame is depicted in Figure 2.

For further detail about each of those subsystems, refer to the respective section.

4.2 Multi-threading model

4.2.1 Threads and their purpose

In the beginning of the project, a threading model had to be chosen. Since the game specification included artificial intelligence for dozens of computer controlled players and realistic physics, the application had to be multi-threaded. Our main concern were uneven computational requirements; later on, profiling tools provided evidence that physics and AI calculation spike in presence of sudden movement or object collisions.

The model is based on the following load estimates:

Module	CPU requirements
Graphics	high
Physics	low - medium
AI	low - medium
Network(dropped)	low
Game logic	low
Input	low

Any module that would possibly require CPU time was assigned its own thread, the rest was joined. The final distribution was decided as such:

"Graphics" thread (main)	Graphics
Physics thread	Physics
AI	AI
"Network" thread	Network, Game logic, Input

4.2.2 Synchronization

Non-trivial time was allocated to shared structures and following discoveries were made:

- Majority of game data is produced by Physics, AI, Network and Input modules. Graphics just consumes data.
- AI, Physics, Network and Input provide more or less exclusive data and do not need to share data members with each other.
- Best approach to local input is sampling (Section 4.8).
- AI, Physics, Network and Graphics can work in parallel.
- Certain non-game-play related actions require messaging.

Due to this and the fear of possible deadlocks and overhead of ad-hoc locking of common structures we decided that one rigorously specified synchronized structure would be the most effective way to solve all problems. Model provides two sets of data. The first set (so called writers) provides an interface for writing and consumers (so called readers) provides an interface for reading. The main idea is that the AI and the physics generate data for the next drawn frame while the graphics module renders the current frame. When all producers finish and a new frame is ready, a swap occurs. Then, the current frame is dropped and data from the new frame are copied into the current frame. Details of each aspect of the synchronization are described in chapters below.

Threading was implemented as a core part of the application and in a time when no other module wad mature enough for integration. It is designed for minimal overhead, while providing messaging and safe access to shared structures. It could be easily removed and used in other module based project.

4.2.3 Messages and heartbeat

All message structures are classes sharing a common ancestor, the `Message` class. Application recognizes two types of messages. `DiscarableMessage` is used for ad-hoc non-periodical events like initialization, loading, game logic events and termination. `PersistentMessage` is used for periodical messages to prevent unnecessary heap operations. Messages are identified by ID. Custom message classes can contain data members.

Physical simulation, graphical rendering, input sampling and communication are all fashioned in a cyclical manner. Every now and then these cycles need to stop and share the generated data. This required barrier synchronization which as in all multi-threaded games is periodical and under optimal conditions constant. We achieved this by making use of the heartbeat pattern. The most idle thread – the "Network" thread – is designated to emit the `HeartBeatMessage` every frame. Upon receiving this message, each producer begins its calculation.

4.2.4 Reader and Writer

The application contains a root synchronization structure called `GameData`. All shared structures are placed inside as members. All members are required to provide the `Bufferable` interface. Reader and writer accessors are provided at the root level of the data structure. `GameData` is a core structure and wraps all data operations. Each operation is executed by locating the target structure, getting reader and writer accessors instances by using the `GetReader` and `GetWriter` methods. Ideally, no data are accessed directly. It is our goal to enforce a common pattern in shared data manipulation. The `GameData` class serves as an adapter to all the shared data. This pattern helped us to manipulate data transparently without worrying about threading problems.

Usage is fashioned accordingly:

1. `HeartBeatMessage` is emitted by the threading core.
2. `GameHeartBeatMessage` is sent to each registered module.
3. Module asks for the Writer interface by calling `StartWork`.
4. Module finishes its calculation and calls `EndWork`.
5. `GameData` recognizes when all modules have finished and there are no readers present, and swaps the frames.
6. Module threads sleep until a new message arrives.

The reader interface can be invoked at any time and the usage of this interface is assumed/designed to be for a short period only. This interface is used by every thread to process inputs and Graphics thread to reflect data into graphics engine.

`HeartBeatMessage` and `GameHeartBeatMessage` are used to control game computation. `HeartBeatMessage` is a core part of the `Thread` class and is localized to one thread only. If the core classes were to be reused, it could used as a standard message-based timer. `GameHeartBeatMessage` is a specialized part of the game.

All modules execute calculations upon receiving this message. It is being emitted only after all modules are loaded. Basically `GameHeartBeatMessage` is a multiplexed version (thread-wise) of `HeartBeatMessage` with frame timing data.

4.2.5 Structures

`GameData` data members use these structures:

- `BufferedValue<T,R,W>` - single structure.
- `BufferedList<T,R,W>` - indexed list.
- `BufferedMap<K,V,R,W>` - ordered map with key lookup.

`R` and `W` point to types used for read and write access to the item itself. `T` and `V` need to provide the `T *Clone()` and `Update(T &_source)` member functions.

Structures are swapped automatically when at the right time. Collections require additional locking (provided transparently by the structure) when new items are added. Adding items is not done directly. New items are appended to an auxiliary list. The next swap joins these items with the collection itself. Collections provide a vector adding function `AddItems` to avoid multiple sequenced locking and unlocking. In the whole game, no two different modules add the same object to a single collection. Therefore we do not have to solve the duplicated keys issue in map collection. Added items are not visible until the next frame.

Deletion is flag based. Items that are no longer needed will have their to-be-deleted flag set. Items stay active until the next swap occurs. During frame swap, flagged items are removed from the structure and deleted.

The mechanism guarantees that collections stay unchanged and correct (from the iterator's point of view) and changes to the collection do not cause thread blocking, ie. iterations over the collection do not need a lock.

Collections provide reader and writer iterators. Generally, all data should be read only from reader iterators and written to write iterators. There are however breaches of this rule is several places. Since it is mostly related to data that do not change during life cycle of an object (reading from a writer), we tolerate these exceptions.

Reader and writer iterator usage example:

```
BufferedList<
  Data::GraphicMenuAction,
  Data::GraphicMenuActionReader,
  Data::GraphicMenuActionWriter
>::ReadIterator action_r = _reader.GraphicMenuActions.Begin();
BufferedList<
  Data::GraphicMenuAction,
  Data::GraphicMenuActionReader,
  Data::GraphicMenuActionWriter
>::WriteIterator action_w = _writer.GraphicMenuActions.Begin();

/* ... */

while(
    action_r != _reader.GraphicMenuActions.End()
&&  action_w != _writer.GraphicMenuActions.End()
)
{
    switch(action_r->Type)
    {
    /* ... */
    }

    action_w.SetToBeDeleted();

    ++action_r;
    ++action_w;
}

/* ... */
```

Structures are strongly typed and implemented using templates. Most of the target structures use helper macros for common implementations of the required interface. Target structure can provide custom reader and writer accessors. When not applicable, plain pointers are provided to access the data.

- `FS_BUFFERABLE` macro - adds plain pointer interface for `GetReader` and `GetWriter`, `Update` uses `operator=`, `Clone` uses copy constructor.
- `FS_BUFFERABLE_EX` macro - `Update` uses `operator=`, `Clone` uses copy constructor.
- no macro - used when `Update` needs custom handling or post processing (see `AirplaneGameFrameEntity`).

It is also worth to mention that classes used in `GameData` members are forward declared. This was done mid-way when game recompilation times became an issue. `GameData` interface remained unchanged, but `R` and `W` template arguments were necessary.

4.2.6 Swap and swap chain

Swap is a barrier synchronization primitive. During a swap only one thread (although it might be a different thread every time) is awake and it is guaranteed that there are no unreleased readers and writers. Routines

executed during the swap have full access to all shared structures. Game modules can register ordered swap hooks during game start-up. They are used for menu and logic actions processing, user input processing, game consoles update and debugging purposes.

Currently the game has the following swap hooks:

- `01input` - polling user keyboard and mouse input.
- `02graphics` - graphical console refreshing, applying menu actions, frame related debug output.
- `03logic` - applying menu events.

Since this mechanism has a very high risk of misuse, we tried to limit the amount of code executed during swap to the minimum. However, it was very useful for identifying concurrency-related bugs. If some new feature caused problems, we could check easily if the problem is thread-related by moving it to the swap hook.

The swap hook is implemented in `Module` subclasses. Each module can provide a virtual function and register the module with a swap hook key into `GameData` structure. Upon every swap, the swap hooks are executed in ascending order by key .

```
void GameLogicModule::SwapHook(const SwapHookKey&, const GameData::Reader
    &_reader, GameData::Writer &_writer)
{
        /* ... */
}
```

A module can be registered multiple times with different swap hook keys and execute different actions depending on the current ordinal position of the swap hook. Both the reader and writer interfaces are fully available.

4.2.7 Messages in detail

Message classes can be divided into abstract (`DiscardableMessage`, `Message`, `ModuleLoadFinishMessage`, `ModuleLoadUpdateMessage`, `ModuleLoadStartMessage`, `PersistentMessage`, `ThreadInitMessage`, `ThreadQuitMessage`) and final groups (the rest).

Each message has its own ID listed in an enumeration in the `Message` class. If message is required to hold additional data, it has to be subclassed. If not, only the registration of a new ID is required. Messages can be sent using the `PostMessage` method of `Thread` or `Module` (thread associated to the module) classes. Message processing starts by overloading the `ProcessMessage` method of the `Thread` derived class. The messages than fall through the modules registered with the thread. The return value of `Module::ProcessMessage` tells you whether the message has been processed. If no-one consumes the message, an assert is triggered. This is considered to be a logical error. In that case, the message has been sent to the wrong thread or module, or its processing not implemented at all.

Sending of discardable messages:

```
graphicsThread.PostMessage(*fs_new ThreadInitMessage());
```

Sending of persistent messages:

```
typedef std::pair<const Module*, GameHeartBeatMessage*> Registration;
typedef std::list<Registration> RegistrationList;

for (RegistrationList::const_iterator reg = this->m_registrations.begin(); reg
    != this->m_registrations.end(); ++reg)
{
    reg->second->UpdateFrameTimeDifference(diff);
    reg->first->PostMessage(*(reg->second));
}
```

The second mechanism is used mainly for loading and game heart beat. However, it is used only sparsely.

4.3 Menu

4.3.1 Concepts

Before the game starts, the player has to configure the game options and career options. This is usually done in the game's menu. According to Internet discussion forums and due to personal recommendations, we chose CEGUI [2] as our menu system. This choice was confirmed by our initial tests, although usability problems appeared at a later development stage.

4.3.2 Implementation

CEGUI subroutines can only be called by the graphical thread. Control logic is centred inside the `GameLogicModule` which runs in the `NetworkThread`.

Since our synchronization model didn't match the requirements of the GUI subsystem, the event model was used. All events are handled by instances of the `GraphicMenuAction` and `LogicMenuAction` classes.

`GraphicMenuAction` handles GUI changes such as layout loading and setting of widget properties. `LogicMenuAction` serializes UI event data to be processed by menu layout handlers. Each widget layout has its own subclass of the `MenuHandler` class and a CEGUI layout file. The class reimplements categorized event handlers, while the base class providers routines for handler registration.

Layout loading and GUI setup takes place in `GameLogicMenu`. CEGUI itself contains a renderer for Ogre3D, but since we patched Ogre3D and broke the binary compatibility, renderer code got included in the project. The renderer still follows the standard interfaces and is not patched.

4.3.3 Handlers and actions

The `Menu` class handles handler registration and wraps CEGUI window objects. Each CEGUI layout has its own `MenuHandler` subclass, which is instantiated upon game initialization in `Menu::LoadLayouts`. The subclass then loads layout resources and, using virtual functions, provides event handler sinks. Each layout window (control) must subscribe to its corresponding events. This is done in the subclass' constructor. Currently, we support the following events:

- `ButtonClicked` handles `CEGUI::PushButton::EventClicked`.
- `TextChanged` handles `CEGUI::Editbox::EventTextChanged`.
- `ListSelectionAccepted` handles `CEGUI::Combobox::EventListSelectionAccepted`.
- `SelectionChanged` handles `CEGUI::Listbox::EventSelectionChanged`.
- `CheckboxChanged` handles `CEGUI::Checkbox::EventCheckStateChanged`.
- `SliderChanged` handles `CEGUI::Slider::EventValueChanged`.
- `ScrollPositionChanged` handles `CEGUI::Scrollbar::EventScrollPositionChanged`.

The GUI is set up and controlled from `GameLogicModule` by `GraphicMenuAction`. This structure holds all the necessary data to configure the desired CEGUI window. All layouts are loaded only once, at start-up, and we use only hiding and activating to alter the GUI appearance as needed. This requires a complete reset of all layout controls, since the previous chain of game states may have left it in an inconsistent state. We support the following actions:

- `HideLayoutAction` - hides current layout, returns display back to game.
- `LoadLayoutAction` - shows layout, hides the old, if any.
- `UpdateProgressBarAction` - sets the data of a `CEGUI::ProgressBar`.
- `ComboboxUpdateAction` - sets the data of a `CEGUI::Combobox`.
- `ListboxClearAction` - clears a `CEGUI::Listbox`.
- `ListboxUpdateAction` - sets the data of a `CEGUI::Listbox`.
- `EditboxUpdateAction` - sets the data of a `CEGUI::Editbox`.
- `CheckboxUpdateAction` - sets the data of a `CEGUI::Checkbox`.
- `SliderUpdateAction` - sets the data of a `CEGUI::Slider`.
- `ScrollPositionUpdateAction` - sets the data of a `CEGUI::Scrollbar`.
- `SetStaticTextAction` - updates text of a `CEGUI:Window`; this property is fairly common and has different uses depending on the subclass.
- `SetVisibilityAction` - hides or shows a `CEGUI:Window`.
- `SetDisabledAction` - enables or disables a `CEGUI:Window`.
- `SetComboSelectionAction` - changes selection of a `CEGUI:Combobox`.
- `MultiColumnListUpdateAction` - load all data for a `CEGUI::MultiColumnList`.

4.4 Game logic

A game's most important quality is it's playability and enjoyability. The means to make a game playable and enjoyable are defined by a *game design document* (which was stored in our DokuWiki, had been lost during the crash of the `urtax` server, and is basically irreproducible); the most immediate implementation of a game design document, in turn, is the *game logic*. Since *game design is not an exact science*, many decisions are based on gut feeling or on a short survey amongst other developers and potential players.

As a game gets implemented, it's design evolves along with it. On one hand, changes that invalidate parts of the architecture have to be made to avoid unexpected gameplay pitfalls, on the other hand, the design has to change to accommodate the project's schedule and avoid features that turned out (as opposed to their perceived difficulty during the design phase) to be too difficult to implement given their gameplay merit.

4.4.1 Logical entities

There are several logical entities within the game (we will refer to them as entities in this chapter) that play the role of various real-life concepts and objects. We believe that terms such as Pilot, Airfield, AirplaneType, Nation, Side (a group of nations), Award (military medal), Rank (military rank) do not need further explanation. Of the other terms, a Squadron is a logical group of pilots, roughly equivalent to, for example, a sports team; a Front defines the geographical position of the front at a point in time; News defines a piece of text that will be shown at some point during the war; finally, a Period defines the properties of a timespan during the war. The relationships between the entities are given in Figure 3.

Some of the entities have parameters that change over time. The front's position or a pilot's assignment to a squadron are good examples.

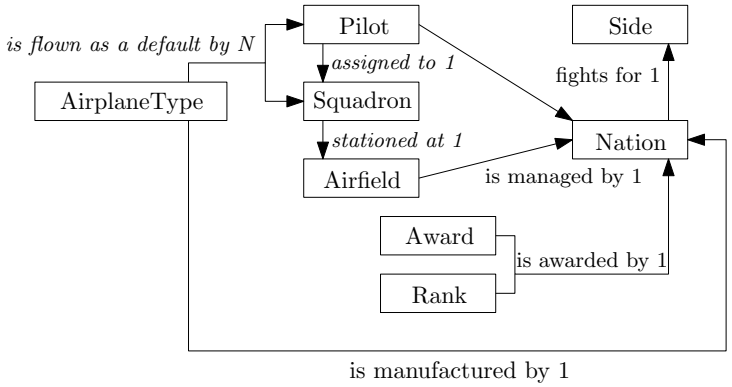


Figure 3: Selected entities and their relationships. Relationships in italics change with time.

There are four airplane roles (see `AirplaneRole`) currently distinguished, namely a fighter airplane, a bomber airplane, a balloon and a Zeppelin; out of these, only the fighters and the bombers are currently present in the game.

Strong Types Because of the large number of various IDs present in the game and the possible bugs that would stem from the eventual mix-up of the various IDs, it has been decided that strong types have to be introduced for some of the IDs, namely those where the meaning of the respective IDs is rather similar, such as the ID of an airplane as opposed to the ID of an airplane's type. Specifically, the following strong types were introduced: `PilotId`, `AirportId`, `SquadronId`, `AirplaneId`, and `AirplaneTypeId`.

Each strong type acts as a wrapper for an integral ID value. It allows for automatic conversion to an integer, but an explicit cast is needed for re-assignment.

Implementation Each entity and each time-dependent relationship is defined in its own class. The 1:1 relationships are usually defined as references within those classes. All of those entities are managed with the `Entities` class, which can be accessed using the `Globals::GetEntites` method. The `Entities` class' calls are re-entrant and thread safe once the class has been initialized. This is due to the fact that the interface provides, with the exception of the initialization code, read-only functions.

Because the `Entities` class acts as the game's a database, a lot of effort has been made to make all the information contained in it easily accessible. The basic getters such as `GetPilot` return a class who's members are cross-references to other classes. For example, getting a pilot's nationality string can easily be done using the following piece of code:

```
std::string pilotsNation(
        Globals::GetEntites().GetPilot(pilotId).Nation.ID
);
```

where `Nation` is a reference to the nation class, and `ID` is the nation class' identification number.

The more difficult queries return a list of results, as does, for example the `GetSquadronsAtDate` function. Each result is then a specialized type containing all the information deemed relevant to the query in question.

4.4.2 Mission

The gameplay is divided into missions. There are five mission types available in the game (see `GameSubType`):

1. **Ace dogfight** - a one-on-one encounter with a famous pilot from the past.
2. **Squadron dogfight** - a squadron vs. squadron encounter.
3. **Bombarder escort** - the player's squadron escorts a squadron of friendly bombers. The goal is to prevent them from being shot down by the enemy squadron.
4. **Bombarder raid** - the player's squadron should primarily stop the enemy bombers and possibly also their escort squadron.
5. **Patrol the front** - similar to the squadron dogfight, the player is supposed to fly along the front and shoot down any enemy airplanes he encounters.

In a mission, one fighter squadron from each side and, depending on mission type, at most one bomber squadron get generated. Before the mission starts, and depending on the mission type, a path for each of the sides is generated and the squadrons are placed near the beginning of those paths.

Each mission's paths are only generated in 2D, being converted to 3D when the airplanes' positions are initialized. The path generation is parametrized by the time relative to the mission's start at which the two sides are to meet.

The airplanes' relative positions are defined by a formation (see `GameFormation`). It is not enough to just generate the positions at the beginning, the relative offsets (see `ComputePathOffset`) to the centre of the formation need to be passed (see the constructor of `AirplaneGameFrameEntity`) to the AI so that it can maintain the formation's shape. There is no other explicit formation information, since only one fighter

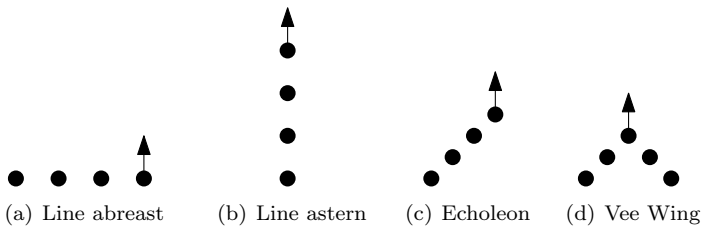


Figure 4: Airplane formations supported by the game. The arrow shows the flight direction of one of the airplanes.

squadron is present for each side and the eventual bomber squadron doesn't engage in combat, therefore never leaves the formation in the first place.

Once the mission starts, the game logic keeps track of the current state of the mission, eventually deciding the ending criteria were met and ending the mission. Depending on the mission type, the ending criteria vary. For example, the bombardier raid mission ends when all enemy bombers have been shot down. Even if the end criteria are met, the player is allowed to choose whether he wants to continue playing the mission or whether to return to the menu (under the condition that there are enemy airplanes left).

The end of the mission is determined by the `HasMissionEnded` function, which returns the `MissionEndInfo` class. The `MissionEndInfo` class encapsulates the mission end information such as whether the mission can continue ("if the end criteria for the mission type have been met, are there still enemy airplanes to shoot down?"), what the cause of the mission's end is (see `MissionEndCause`), etc. Because of the Reader/Writer paradigm, the information is propagated in the `Mission::m_endInfo` member variable (set using `SetMissionEndInfo` and queried using `GetMissionEndInfo`).

While there are several pre-defined mission end causes (see the `MissionEndCause` enumeration), the currently used ones are `MissionEndSuccess` (mission criteria have been met), `MissionEndFailure`, `MissionEndTime` (mission time ran out), `MissionEndAbandoned` (the player abandoned the mission), and `MissionEndDeath` (the player died).

Three stages of death are recognized (see Figure 3). The first one (see `RegisterPlayerWillDie`) means that the pilot is bound to die sooner or later; aside from the obvious `PilotId` of the pilot who "will die" soon, the function takes an optional parameter specifying the killer's `PilotId`. The second stage of death (see `RegisterPlayerHasDied`) specifies that the player has already died, usually by hitting the ground too hard or by getting hit by a bullet directly. The final stage of death is when the player's wreck gets deleted from the game (see `RegisterWreckRemoved`).

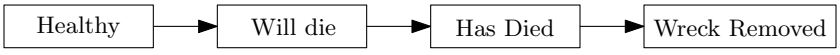


Figure 5: The three stages of death and their ordering. Note that the healthy stage is not explicitly represented in the game.

The above mentioned methods are called by the physics module. It is assumed that if one of them is called, all the preceding stages' methods have already been called. Without calling the `RegisterPlayerWillDie`, no information about who shot down which airplane would exist. But should this callback be removed, an airplane could be heading uncontrollably towards the ground, implicating the pilot will get killed in the process. Yet if they player quit the game before the impact happened, the kill would not get registered. If the player dies, we want the user to be able to view the smoking wreckage for a short while before going

back to the menu. Therefore, we schedule the (human) player's wreck to get removed much sooner than the AI wrecks, ending the game once it has been removed, hence the need to register the wreck removal (`RegisterWreckRemoved`). But since the AI wrecks remain on the ground for several minutes, it's not practical to use this information to count the number of remaining enemies. Therefore, `RegisterPlayerHasDied` had to be introduced. Once all AI pilots "have died", the mission can safely end, and the eventual shock-menu effect (caused by using the "will die" level, which gets triggered as soon as the AI is bound to die) as well as the no-menu effect (caused by the logic waiting for all AI wrecks to disappear) is avoided.

Score The missions score is calculated (see `GetMissionScore`) using the following formula:

$$s = c \cdot (10 \cdot N_{fighters} + 8 \cdot N_{bombers})$$

where s is the final score, $N_{fighters}$ is the number of fighter airplanes the user shot down, $N_{bombers}$ is the number of bomber airplanes the user shot down, and

$$c = \begin{cases} 0.7 & \text{mission's goals weren't achieved} \\ 1.6 & \text{mission's goals were achieved} \end{cases}$$

This score model is rather simple. It could be greatly improved by taking the total number of shot down enemy airplanes and other statistics into account.

4.4.3 Career

The career mode is a a game mode built over the regular mission. By adding a sense of continuity and the ability to develop his/her own career, the user should get more engaged in the game than in the single mission mode.

Military ranks and military awards are one of the player's main motivations. A record of the user's achievements is kept throughout the career. The more successful the user gets, the higher rank he'll attain (`AttainedRank`) and the more prestigious awards he'll receive (checked in `OpenDebriefing`).

The career mode keeps track of current date (`m_date` member in `Career` class, where the whole career mode is implemented) to know where between the career mode's beginning and end (see Section B.1) the user currently is. After each mission, as well as at the beginning of each career, the number of days to elapse until next mission and the next mission's parameters (type, formation, number of friendly and enemy airplanes, etc.; see `PlanNextMission`) are generated.

A list of famous pilots (aces), along with their dates of deaths, is loaded into the entities. Some of them died during the war, and some of them might have been shot down by the user. A list of aces that are still alive is therefore kept (in `m_acesAlive`) and updated after each mission, deleting both aces that have been shot down in the game as well as the aces that during that period of time. Those will no longer be scheduled to fly mission with or against the player.

The user is always assigned to one of the squadrons. Depending on his rank, he can either be transferred without his own consent, or request transfer to one of the other squadrons himself. The squadrons' assignment to airfields, as well as airfield position, can (and if defined so in the datafile, will) change over time.

4.5 Graphics

The graphical subsystem is an important part of any computer game. It is responsible for the visual presentation and appearance of the game environment. Graphics, along with an intriguing gameplay, is the most significant factor that attracts players' attention. Since our game has been planned as a realistic combat flight simulator, we have tried to make it look as realistically as possible with the limited resources we had at our disposal.

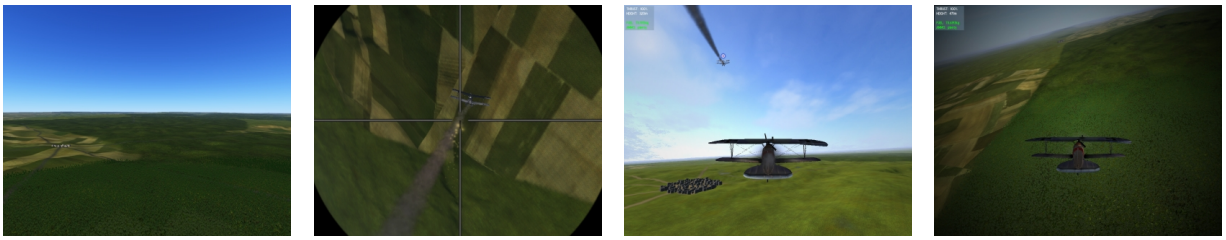


Figure 6: Screenshots from Flying Samurai.

Figure 6 lists a few screenshots from the game. Some of the graphical effects included in the game can be seen there, such as terrain rendering using texture splatting, large forests, approximative sky scattering, particle effects such as smoke and bullets, clouds (unfortunately not volumetric ones), or fullscreen effects like HDR rendering, tunnel vision and aiming cross-hair.

The cornerstone of our graphical subsystem is the Ogre3D graphics engine [19]. Ogre3D is an open source project, that is quite established in its large community, which was the main reason why we chose it. From a technical point of view, its architecture is quite well thought through and it is very stable, considering it is an open source project. The main problem of Ogre3D is the fact that it is a general graphical engine, usable a for wide spectrum of 3D applications ranging from games to 3D editors. This is actually a slight downside for us, as it can't match a narrowly specialized game engine in terms of performance. But this is something we weren't even expecting from it. There is also quite a number of plugins for Ogre3D, which enhance its capabilities. From these, we are using Myrddin Landscape Plugin [18] (MLP) for terrain rendering (see Section 4.5.3 for details), PagedGeometry Engine [22] for rendering of large batches of static geometry (see Section 4.5.4), HDRlib plugin [11] which provides HDR rendering capabilities and CEGUI library [2] for displaying GUI components. Since these plugins are developed by Ogre3D users who haven't participated on its core development, the stability of these plugins is usually not as good anymore. This has been problematic mainly for MLP (partly because it's still in its beta version) and CEGUI, and not so much for the other two.

The graphical subsystem in Flying Samurai is accessed through the`GraphicsModule` class, which is also its core part. Since graphical subsystem is a tightly grouped part of the game, there is a quite low number of other classes associated with the `GraphicsModule` class. The workflow of `GraphicsModule`'s functionality has two major parts – translation of shared game structures into graphical representation and rendering (display) of this representation. At the beginning of each frame, `GraphicsModule` locks and reads (see Section 4.2) the shared internal game structures which contain information about game entities and updates their graphical representation accordingly. Then Ogre3D's rendering pipeline is invoked and one graphical frame is rendered based on this graphical representation of the game scene.

The following sections describe the most significant aspects of Flying Samurai's graphical subsystem: Section 4.5.1 describes the structure of the scene graph, Section 4.5.2 contains the description of airplane meshes, Section 4.5.3 describes how the terrain is modelled and rendered, Section 4.5.4 explains the structure of classes that handle static geometry in the game, Section 4.5.5 explains the functionality and structure

of the in-game HUD, Section 4.5.6 mentions some special effects in the game, Section 4.5.7 says a few words about camera control in Flying Samurai and Section 4.5.8 briefly describes our GUID system.

4.5.1 Scene graph

Scene graph is a basal data structure which represents an abstract structure of the scene and the hierarchical relations of objects in it. Scene graph is composed of nodes which form a tree; each node can have an arbitrary number of children. Some of the nodes also attach various movable objects called 'entities' (according to Ogre3D's terminology – these are not the same entities as logical entities described in Section 4.4.1). These represent renderable objects in the scene. Each node (not just leaf nodes) can have one or more entities attached to itself. The hierarchy in the scene has not only some abstract purpose, but also represents a hierarchy of objects transformations, which are concatenated during the descent in the scene graph. Each node has a certain position in the world space, which also determines the position of the attached entities (where they will be rendered). When the scene is about to be rendered, the Ogre3D engine traverses the scene graph and renders those parts of it, which are currently visible to the camera (visibility culling) and are not marked as invisible (which is possible and reasonable in some situations).

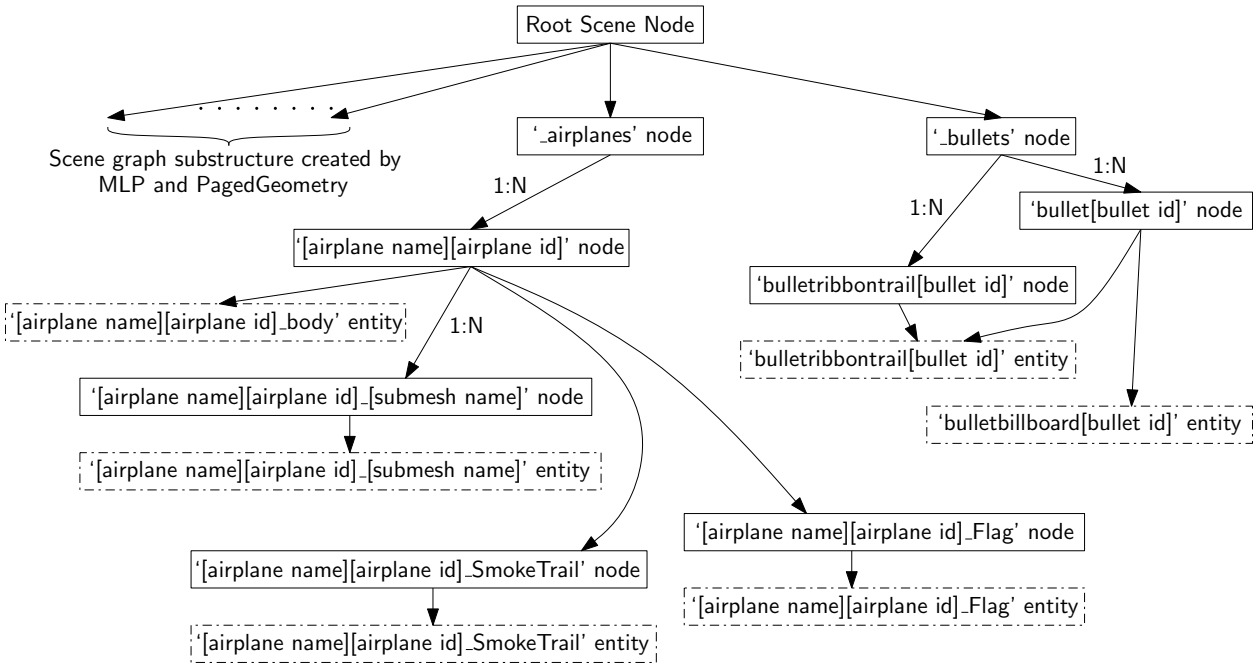


Figure 7: Structure of the main scene graph.

Our scene in Flying Samurai contains four basic object types (except for some auxiliary geometry). These are terrain tiles (Section 4.5.3), static scene geometry (Section 4.5.4), airplanes and bullets. See Figure 7 for the schematic depiction of the scene graph structure.

Terrain renderables and static scene geometry (trees and houses) are managed by third party Ogre3D plugins (namely Myrddin Landscape Plugin [18] and PagedGeometry Engine [22]). These create and manage their own scene graph structure attached to the root scene node. We won't go into detail here, because we don't have any control over this part of the scene graph.

Airplanes are the key objects in our scene. For these we have a special `'_airplanes'` node, to which all active airplanes are attached (so we can easily iterate over them). Every airplane has its own node marked by its unique ID (as all nodes in the scene graph must have unique names). To this node the main `'_body'` submesh is attached, as it has zero translation in respect to the airplane's position. The rest of the airplane's submeshes (see Section 4.5.2 for the enumeration of all submeshes the airplane can have) is attached to the airplane's body via their respective nodes suffixed `'_[submesh name]'`. The reason why certain parts of the airplane are decoupled from its body is either they need to be animated (ailerons, propeller etc.) or because the physics module needs to load them separately for physical simulation (such as wings or pilot cage). Two more nodes are attached to the main airplane node and these are the `'_SmokeTrail'` node/entity which represent the smoke system activated when the airplane is damaged and the `'_Flag'` node/entity, which represent an overhead sign that marks which side of the conflict a particular airplane belongs to.

The last group of objects are bullets. They again have their own `'_bullets'` node for easy iteration over them. A bullet is represented by 2 nodes – its main `'bullet'` node marked by an unique bullet ID and the `'bulletribbontrail'` node. The main node has a flare billboard marking the bullet position attached and along with the second node, they also have the bullet ribbon trail attached (which represents a smoke trace behind the bullet and has to be attached to two scene nodes).

4.5.2 Airplane meshes

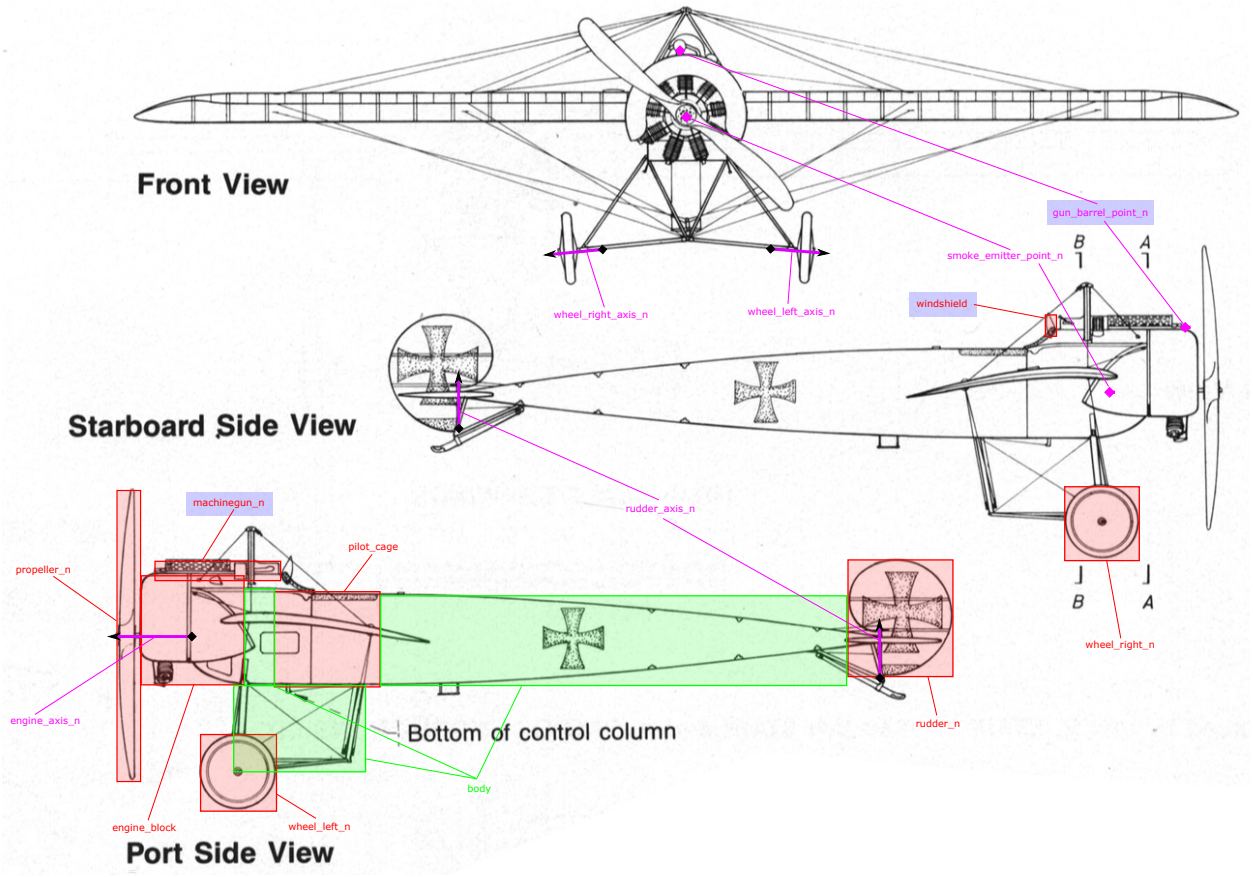


Figure 8: Subdivision scheme of an airplane mesh, part one.

Airplanes are graphically represented by mesh models which are attached to their corresponding scene nodes as described in Section 4.5.1. These meshes are further subdivided into smaller submeshes, for two reasons: either they need to be animated or they are important or vulnerable parts of an airplane and thus have to be loaded separately by the physical module (or both). This section describes general rules about how an airplane model should be subdivided into these submeshes and their naming conventions (which is sort of overlapping with Section F, but conceptually it belongs here, because there's a direct connection between scene graph structure and airplane submeshes).

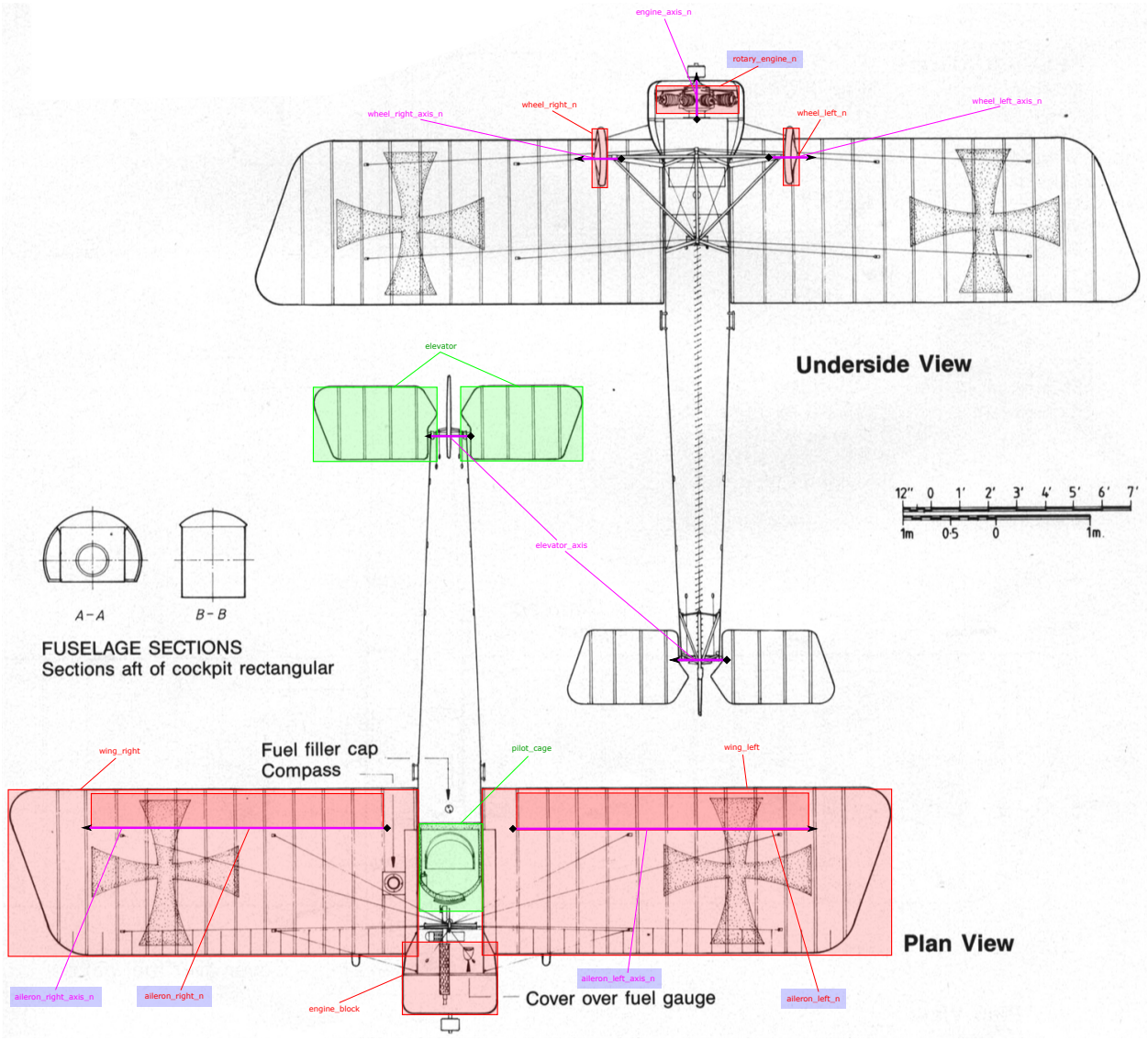


Figure 9: Subdivision scheme of an airplane mesh, part two.

Figures 8 and 9 show an example airplane (Fokker E III) with all submeshes marked by coloured boxes and their corresponding labels. Colours of the boxes don't have any meaning, they are there just for better distinction. Purple labels mark invisible auxiliary geometry, as explained below. If a label has a light-blue background, it means that the corresponding mesh is not present on all airplanes (e.g. not all airplanes were actually armed with a machinegun). If a label contains suffix `'_n'`, it means that there can be more

instances of that mesh type on a particular airplane (e.g. some airplanes had more than one machinegun) – the `n` is then an integer value, starting from `0`. Some submeshes don't have this suffix, even though there can be more of them, but in that case they can be joined into one submesh, because they don't need to be animated and the physical module doesn't need to distinguish between them (for instance, there can be more engine blocks, but the physics don't need to know which one was hit, just that any of them was). The names on the labels correspond only to the `[submesh name]` part in Figure 7, the submesh files in the respective airplane directory have to be moreover prefixed with `[ariplane name]_`.

The data which describe one complete airplane are these: a set of `.mesh` files, which represent their respective submeshes, three textures in `.dds` format (see Section F) and one `.scene` file. This file is generated by a `.mesh` file exporter and describes relative spatial positions of all submeshes in local coordinate system. The format is described in [6], but we utilize only the `name`, `position` and `scale` attributes. The `GraphicsModule` class loads and reads this file manually (i.e. we don't use any of the available plugins for it). All subnodes of the main airplane node are positioned based on the values in this file.

Normal meshes Normal meshes are visible renderable solid parts of an airplane.

- `aileron_left_n` (not present on all airplanes, animated)
  Flight control part, controls airplane rolling. Each left wing can have one, but some older aircrafts used different methods for rolling ('wing shearing').
- `aileron_right_n` (not present on all airplanes, animated)
  Same as the previous, but on the right wings.
- `body` (present on all airplanes)
  Root airplane mesh, always has zero offset in local coordinate system and is attached to the main airplane node.
- `elevator` (present on all airplanes, animated)
  Flight control part, controls airplane pitching.
- `engine_block` (present on all airplanes, special collision body)
  Encapsulates airplane engine(s). More vulnerable than the most of the airplane.
- `machinegun_n` (not present on all airplanes, special collision body)
  Airplane's primary weapon. More vulnerable than the most of the airplane (but there's a very low probability of hitting it).
- `pilot` (present on all airplanes, special collision body)
  Pilot's body, not drawn on the schemes. Extremely vulnerable.
- `pilot_cage` (present on all airplanes, special collision body)
  Basically the outer cockpit bounds. Very vulnerable, because it is easily penetrated by bullets, which then reach the pilot.
- `propeller_n` (present on all airplanes, animated)
  Airplane's propelling device. There can be more engines one a single airplane, and therefore also more propellers.
- `rotary_engine_n` (not present on all airplanes, animated)
  Rotary engines were special types of engines, which rotated on a fixed shaft with the same angular speed as the propeller (for sake of an efficient cooling). This allowed an engine to be much lighter, as it didn't need a heatsink. However, this caused the airplane to constantly roll to one side, so the pilot had to compensate for it.

- `rudder_n` (present on all airplanes, animated)
  Flight control part, controls airplane yawing. There can be more of them for higher yawing performance.
- `wheel_left_n` (present on all airplanes, animated)
  Wheel which is in contact with the ground during takeoff and landing.
- `wheel_right_n` (present on all airplanes, animated)
  Same as the previous, but on the right side of the fuselage.
- `windshield` (not present on all airplanes, different material)
  Cockpit windshield, not every airplane had one.
- `wing_left` (present on all airplanes, special collision body)
  Joined left wings (some places had 2 or 3 main wings). Susceptible to damage, which can change the airplane lift, causing it to roll to one side.
- `wing_right` (present on all airplanes, special collision body)
  Same as the previous, but on the right side of the airplane.

Auxiliary meshes Auxiliary meshes, as the name suggests, serve as helper objects to the normal meshes. Apart from the normal meshes, these are neither visible, nor are they part of the scene graph. We have two kinds of auxiliary meshes: axes and points. Axes serve as anchors for their respective submeshes, which rotate around them during the animation. We chose this representation of submesh rotation axis over utilization of the `quaternion` attribute in the `.scene` file, because it's more intuitive for artists to place and align these additional meshes. Points mark some important places on the airplane model.

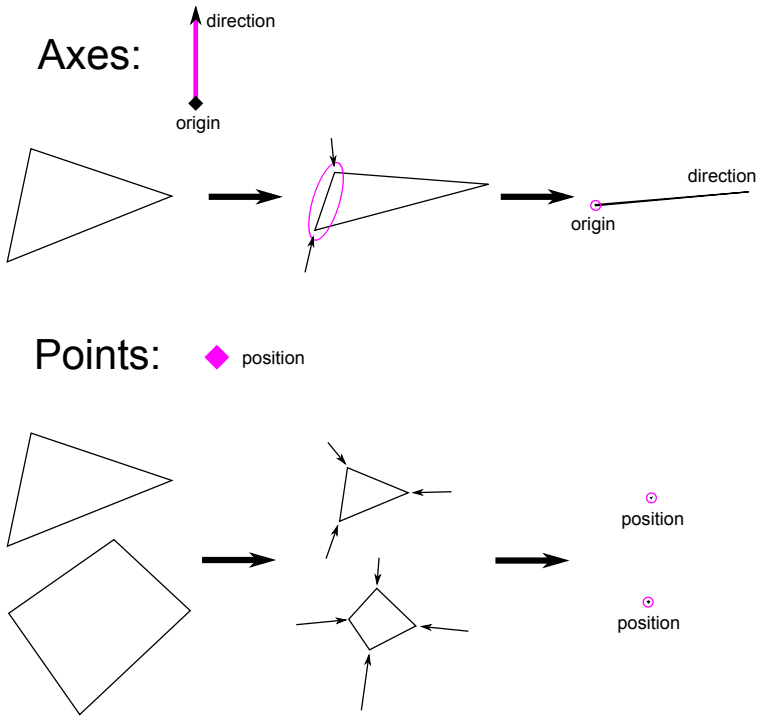


Figure 10: Proxy geometry representation scheme.

The construction of these meta-objects is very simple, and is sketched on Figure 10. Axes are constructed by placing a triangle into the scene in a modelling application, and two of the points are collapsing onto the

same place. Now they form a vector – the game detects the two collapsed points as being next to each other and marks them as the origin of the vector; the third point then determines the direction of this vector. This vector is then normalized and serves as a rotation axis for the respective submesh. It's important not to misplace them during the modelling, as the rotation won't behave well anymore. It's also necessary that they are oriented as shown on Figure 10, otherwise the rotation will be reversed. Points are constructed similarly – all vertices of a triangle or quad are collapsed and this collapsed primitive is then placed onto the intended place on the airplane model.

- `aileron_left_axis_n` (not present on all airplanes)
  Left aileron rotation axis. There can be more of them, as there can be more left ailerons on the same airplane.
- `aileron_right_axis_n` (not present on all airplanes)
  Same as the previous, only on the right wing.
- `elevator_axis` (present on all airplanes)
  Elevator rotation axis.
- `engine_axis_n` (present on all airplanes)
  Rotation axis for the propeller and for the rotary engine, if present.
- `gun_barrel_point_n` (not present on all airplanes)
  This point marks the tip of its corresponding machinegun; the game uses this point to initialize newly shot bullet's position.
- `rudder_axis_n` (present on all airplanes)
  Rudder rotation axis.
- `smoke_emitter_point_n` (present on all airplanes)
  Placed into the centre of an engine, this point marks the position where the smoke system indicating airplane damage is placed.
- `wheel_left_axis_n` (present on all airplanes)
  Left wheel rotation axis.
- `wheel_right_axis_n` (present on all airplanes)
  Right wheel rotation axis.

4.5.3 Terrain

The terrain is one of the most significant parts of the graphical content in our game. In order to cover the entire area involved in the aerospace combat during the World War I. (see Figure 11), we use data spanning roughly an area of 450km by 450km. These data include terrain geometry, splatting maps and textures, trees and houses. For the description of the workflow for creating and processing these data, and getting them into the game engine, please refer to Appendix D.

Terrain geometry The terrain is represented by heightmaps, which is one of the standard methods to represent terrain geometry (see [12] for instance). We use realistic heightmap data for the involved region obtained from Google Earth [9]. These heightmap data are divided into an array of 32×32 tiles (pages), and each of the tiles has the resolution of 513^2. Given the size of the region, this is equivalent to approximately one heightmap sample per 25 meters, which is more than sufficient for use in a flight simulator, where the terrain is viewed mostly from distance. The heightmap tiles are stored as little endian .raw files (named `hf_hm_[x]_[y].raw`) using 16 bits per sample.

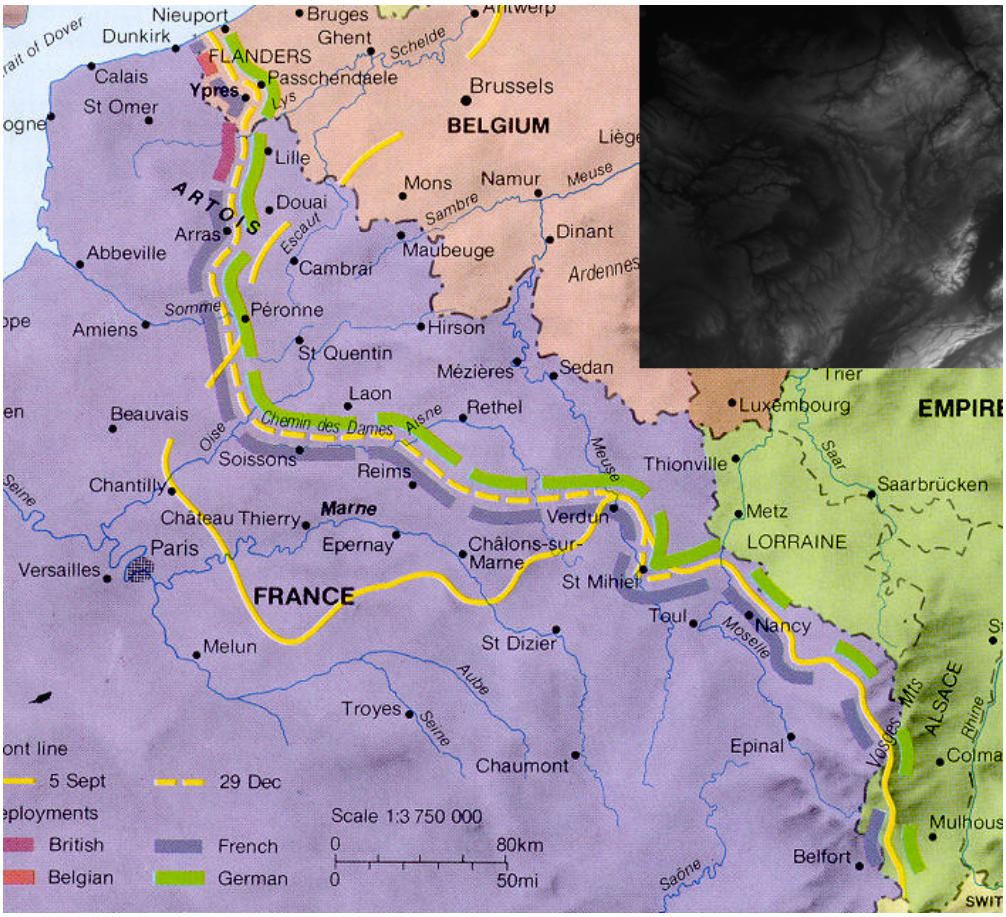


Figure 11: The area spanned by our terrain. In the upper right corner, the corresponding downscaled heightmap can be seen.

Splatting data Splatting [1, 8] is a technique to texture and colour virtual terrains, and is especially suitable for large terrains, where storing high resolution textures for the entire terrain would easily lead to tens of gigabytes of texture data. In short, splatting uses splatting maps (masks) with relatively small resolution, where each of the masking layer corresponds to one type of surface material and then a set of several per-material textures, which are tiled across the entire terrain and their contribution is controlled by the splatting masks. Thanks to the interpolation between the layers and between samples, the transitions between different materials are smooth, and the use of tiled textures provides the terrain with details. Figure 12 shows a screenshot from our game, showing the terrain textured by splatting.

We use 32×32 splatting maps (so there's a 1:1 correspondence with the heightmap tiles), each having the resolution of 1025^2. Each tile is represented by two 4-channel .tga files (named `hf_sm[i]_[x]_[y].tga`, where `[i]` is either `0` or `1`); each of the eight channels corresponds to one type of surface material, except the last one, which we don't utilize. Therefore we operate with 7 splatting layers, which correspond to the following types of surfaces (the correspondence to the concrete channels in the splatting textures is marked by L0 and L1 (the first and the second file from the pair) and R, G, B and A (red, green, blue and alpha channel in the files)): 2 different types of grass (L0R, L0G), rock (L0B), forest (L1G), snow (L0A), field (L1B) and roads (L1R). These data are generated from the heightmaps, partially by the L3DT software [16] and partially by our own tool CityGen (see Appendices D and I for the description of CityGen and the

Figure 12: Splatting in Flying Samurai (grass, rock, field and road layers are present).

terrain generation pipeline).

Trees and houses These are static objects placed on the terrain to enhance richness of the game environment. They are thoroughly described in Section 4.5.4.

For rendering the terrain we use the Myrddin Landscape Plugin [18] (MLP) for Ogre3D engine. It is an implementation of `SceneManager` class of Ogre3D designed for rendering large terrains. It implements paging and Level-of-Detail capabilities, which are both essential for rendering terrains of sizes comparable to ours. However, in spite of being designed for large terrains, the author haven't designed it for terrains spanning several hundred of kilometers, and as a consequence of that we had problems with numerical accuracy, for instance from close views the terrain is slightly shaking. The stability of this plugin is also not very good, as it is currently only in the beta version and the author has abandoned the project some time ago. Integration of MLP into Ogre3D was not very smooth too, mainly because of its complicated configuration. Still, we kept using it, because it is the only viable terrain rendering plugin for Ogre3D engine (we have also tried Paging Landscape Manager 2, which performed terribly and had severe stability problems. We have also considered writing our own terrain rendering system, but implementation of all capabilities we need would take much more time, than integration of MLP and overcoming of the problems with it).

MLP is capable of paged loading of the terrain elevation and splatting data, and of rendering the terrain based on these data in real-time. It also provides us with some miscellaneous features, such as approximative sky scattering computation (without clouds), fog rendering and animation of the moving Sun as a directional light source. All its functionality is configured from the `land.xml` settings file.

It's good to mention that we don't model water surfaces in Flying Samurai. There are multiple reasons for that. First, we use real terrain data ranging in their altitudes from 0 meters at the seashore up to about 2000 meters in Alpes around Switzerland. This means that the riverbeds are sloped and the lakes would

have their water levels in different altitudes, just like in reality. This however is very problematic to tackle with the currently used methods to model large water bodies (which are in most cases modelled as a single large water plane at the ocean level) and the MLP plugin don't support such complicated water bodies (only the simple single-plane approach). Second, despite our geometry being fine enough for terrain in a flight simulator, it is still too coarse to account for the shape of most riverbeds, and MLP does not contain any constraints to mark the rivers, which leads to the problem that rivers are washed out in distance by Level-of-Detail mechanisms. Since player is focused mostly on the combat during the gameplay, we decided not to include water surfaces in Flying Samurai.

4.5.4 Static terrain geometry

Static scene geometry comprises objects that are positioned on the terrain and enhance its richness. Flying Samurai contains two types of static objects – trees and buildings. Rendering these objects is not an easy task (we have roughly a billion trees and up to a million houses spread over the terrain). For this purpose we use Paged Geometry Engine (hereinafter PGE) plugin for Ogre3D, which uses advanced techniques such as geometry paging and instancing to render large portions of static objects. We found PGE to be not only stable and fast, but also relatively easy to integrate into the game engine. It saved us a lot of time and we can only recommend it.

Trees Trees are grouped into forests, naturally. Of course, we can't store positions of each individual tree, because this would occupy several gigabytes of HDD space at best, considering the number of trees we are working with. Instead, we use density maps to store local densities of the forests (these maps are generated from the splatting maps, from the layer L1G that corresponds to forests, see Section 4.5.3). We again have an array of 32×32 tiles (named `tile_[x]_[y].tga`), so each tile corresponds to one heightmap tile, similar to the splatting maps. Each tile is a single-channel .tga file with the resolution of 1024^2. Each texel of this map is an 8-bit value ranging from 0 to 1, and corresponds to a particular area with the size of 14×14 meters on the terrain. The value of 0 means no trees are present on the area corresponding to that particular texel, and the value of 1 means full tree density for that area. The trees are then randomly generated over this area and this number is determined simply by ⟨texel value⟩ · (`MaxForestDensity`+1), where the value of `MaxForestDensity` is obtained from the graphics.cfg configuration file, and is further configurable from the in-game graphics options.

This approach has some intrinsic limitations. Firstly, it's not possible to straightforwardly include more types of trees in the game, since we store only the forest densities. This could be remedied by using density maps with more than one channel or by determining the forest type according to the terrain elevation. However, we don't need to do this, because due to the lack of artistic resources, we have only a single tree at our disposal, which we randomly scale during the positioning on the terrain to include at least slight variation to the uniformity of our forests. Secondly, it's not possible to have solitary trees (for example tree alleys along the roads) with this approach. This could be done by having a separate storage for these trees, where the exact positions of such trees would be stored (this is feasible, as the amount of these trees would be only a fraction of all trees on the terrain). However, due to the lack of time and limited capabilities of our content generation tool, we don't currently do this.

The trees themselves are geometrically represented by a single billboard covered with a tree texture. This is necessary, because the amount of trees in a single frame can reach hundreds of thousands on the highest settings, which would lead to a severe performance hit if we'd used even a very rudimentary geometrical representation for each tree. This representation is indeed very rough, but since forests are mostly viewed from distance, it doesn't decrease the quality of their appearance in most situations.

Buildings Buildings are the basic building blocks of villages and towns. We generate them with our CityGen tool (see Appendix I). CityGen generates locations of the buildings on the terrain and their rotations, along with an unique ID, which is understood by the engine and a proper building is placed on the given position based on this identifier. The list of the generated buildings is then divided into a set of binary files by our TileCombiner tool (see Appendix D). These files again have a 1:1 correspondence to the heightmap tiles and contain binary data about the buildings on that particular tile. These are read by the game engine, which positions the corresponding types of buildings on the terrain in runtime.

Structure of these files is very simple. CityGen produces a single ASCII text file containing the following four-tuples:

```
[<building ID> <position X> <position Y> <rotation>]
```

Each of these four-tuples corresponds to a single building. The TileCombiner tool then creates the array of 32×32 files (named `citytile_[x]_[y].bin`), which contain the same data, but are binary. The quaternions are serialized as 16-bytes long sequences, there the first 4 bytes are an `unsigned int` representing the `<building ID>` and the next 12 bytes are three `float`s, which represent the spatial coordinates of the building.

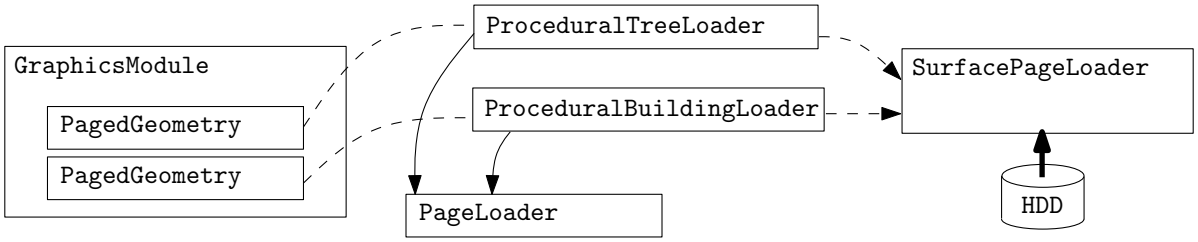


Figure 13: Scheme of the static geometry subsystem.

The subsystem that handles the generation of the static geometry on-the-fly and its rendering is sketched in Figure 13. The graphical subsystem `GraphicsModule` (also see Section 4.5) contains two instances of `PagedGeometry` class – `m_pagedTreeGeometry` and `m_pagedBuildingGeometry` (each of them handles its respective type of geometry). The `PagedGeometry` class is a core class of PGE and provides access to the geometry paging features. It also communicates with the instance of Ogre3D's `SceneManager`. These two objects then query the procedural loader classes everytime a new page of the geometry is needed (each of them uses a different size of the geometry page, determined from the graphics.cfg settings file). This is done by calling `ProceduralTreeLoader::loadPage()` and `ProceduralBuildingLoader::loadPage()` callback methods by the two `PagedGeometry` objects. The `ProceduralTreeLoader` and `ProceduralBuildingLoader` classes inherit from the PGE's abstract `PageLoader` class and perform only the generation (positioning) of the geometry on the terrain. The data for this are provided by the `SurfacePageLoader` class, which is a simple paging microengine, that loads the pages of the static geometry data from HDD on demand (by calling `SurfacePageLoader::GetPageAt()`) and stores them in a small FIFO cache. These data include the tree density maps, the city tiles, but also the heightmaps, so the trees and the buildings are placed at the correct altitude on the terrain.

This design based on lazy evaluation works well; the two `PagedGeometry` objects query the data when needed, always within a certain perimeter around the current position of the camera in the scene. They deallocate the geometry pages automatically when they get outside this perimeter. They also perform visibility culling. The paging mechanisms of the `SurfacePageLoader` class ensure that the pages are not loaded from HDD repeatedly each frame, and the small size of the cache ensures that the memory consumption of this subsystem won't be too high (we cache 10 pages and each page has the size of roughly 1.5MB with the currenty used

resolutions).

A building itself is graphically represented by a single `.mesh` file and two textures. The mesh should have as low amount of polygons as possible and the local coordinates origin should lie in the middle of the bottom side of the building. All details regarding the building models are explained in Sections F and E.

4.5.5 HUD and debugging graphics

HUD – head-up display – is (almost always) a screen-aligned piece of graphics that informs the player about the game's status or statistics. The HUD in our game is rather simplistic, but it serves it's purpose well.

All of the HUD gets shown and hidden depending on the gamestate (parts of the HUD can be hidden on demand; in the menu, no HUD is shown) and whether the aiming cross is activated. Any quote-delimited strings that appear in this subsection are object or node names.

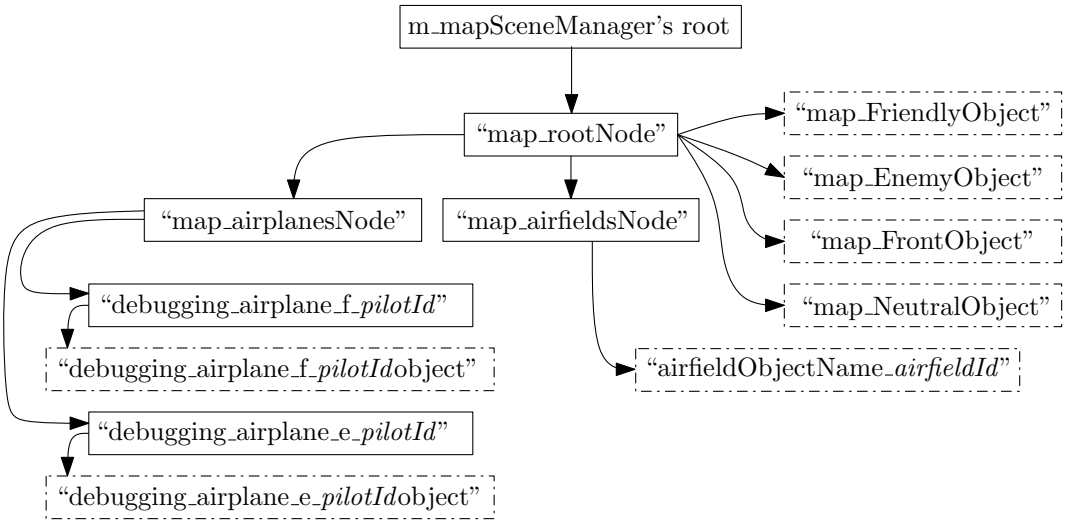


Figure 14: Structure of the scenegraph with respect to HUD and debugging output. Scene nodes are in normal boxes, objects are in dash-dotted boxes, root is in a heavy-framed box. Substrings in italics are substituted with respective numerical values; each such object actually represents a group of objects. Note that, while the names given here are strings, constant variables are actually used instead.

Map The map geometry gets rendered in a separate viewport (see `m_mapViewport`), which allows for easier positioning and clipping. The geometry is placed in a direct sub-node of `m_mapSceneManager`'s scenegraph (see `m_mapName`) (see Figure 14).

The respective part of the scenegraph gets reinitialized (see `InitializeDebuggingNode`, `InitializeMap()`, `InitializeHUD`) with each new mission. In each frame, a new centre of the map is calculated and according to it, all of the objects are regenerated (see `RefreshMapAirplanes`, `RefreshMap`, `RefreshMapAirfields`).

On the map, the friendly path (see Section 4.4.2), the enemy path, the front, and the map's and viewport's frame are shown (the map's frame is only visible with some map-scales; see Figure 14). Each of those objects has a simple custom material (see `InitializeMap`). The airplanes and airfields also have custom materials (defined in `Media\debugging\debugging.material`).

Map centre Map centre – the point around which the map's display will be centred – is calculated separately for each dimension. First, the side s of the visible portion of the displayed map is computed. Then, a border of the $s/2$ size is stripped off the map, the remaining region being called the core area (see Figure 15). As long as the user's airplane stays within this core area, the visible area of the map will fully overlap the map's viewport. If the airplane is in the the border strip, the map has to stop moving along with the airplane, or else parts of the viewport won't be covered by the map anymore.

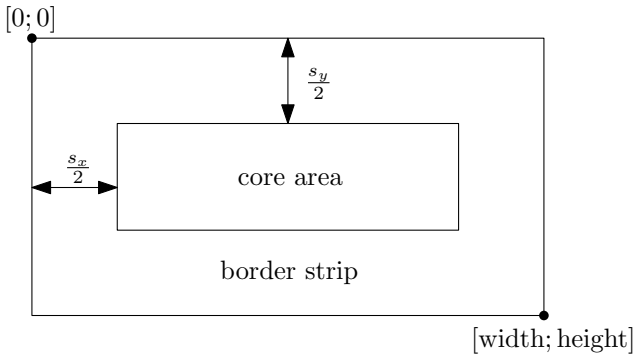


Figure 15: Illustration of some of the values needed for map centre calculation.

When the airplane is in the border strip, it no longer gets drawn in the middle of the screen. Because the map viewport is centred in the screen and its display is positioned based on the calculated map centre, no special logic is needed to display the user's airplane differently. As with all other airplanes, only the offset with respect to the map display's calculated centre (which aligns with the screen centre) needs to be calculated.

Rest of the HUD The airplane control's part of the HUD is displayed in an Ogre3D's `Overlay`. A `PanelOverlayElement` is then used as a logical and graphical container for the respective texts, represented by `TextAreaOverlayElement`s.

3D paths To the main scenegraph's root, a subnode called "debugging_node" is attached. Under it, two separate objects, namely "debugging_pathsFriendlyObject" and "debugging_pathsEnemyObject", are attached. They represent the 3D paths (see Section 4.4.2) the squadrons should follow. Those are exclusively used for debugging purposes and cannot be enabled in the public release version of the game.

4.5.6 Special effects

We have some nice special effects in the game, for example an advanced local reflectance model for airplanes, particle effects like smoke and bullets, or fullscreen compositors like HDR and tunnel vision.

We use Strauss local reflectance model [27] for airplanes shading. This is a very nice model, and while it is empirical, it gives much more realistic results than the standardly used Blinn-Phong model. We also use smoothness and metalness maps to modulate some of its parameters across the airplane surface (see Section F).

For visualization of bullets and smoke, we use particle systems available in the Ogre3D engine. Smoke (which is active when an airplane is damaged) is modelled as a particle system with our own custom material,

emitter and affectors. Its density is derived from the current airplane damage, and is calculated in the `GraphicsModule::UpdateSceneGraph()` method. Bullets are modelled as billboards with 'smoke' ribbon trails attached to them.

Fullscreen post-processing effects enhance the perceived visual quality of the game. HDR (High Dynamic Range) rendering allows rendering of scenes with high dynamic range of illumination in them (such as the sky). For this, we use the HDRlib [11] plugin into Ogre3D, which provides us not only HDR, but also a fullscreen bloom effect. Tunnel vision is an effect of darkening of the peripheral areas of human vision, when subjected to high acceleration. We model this effect by multiplying the rendered image by a radial gradient texture, which darkens the edge areas of the screen. This multiplication is weighted by an acceleration coefficient, which is again computed in `GraphicsModule::UpdateSceneGraph()`. Also, the aiming crosshair (invoked by pressing SPACE during a mission) is modelled as a fullscreen effect by modulating the rendered image with the crosshair texture. All fullscreen effects in Flying Samurai are modelled using Ogre3D's `Compositor` pipeline.

4.5.7 Camera

Camera is an important part of the graphics pipeline, as it represents the 'eye' of the player in the game. In Flying Samurai the camera is represented by the `CustomCamera` class. This class is a relatively small extension of the default Ogre3D's `Camera` class. We chose encapsulation of the `Camera` class over its specialization – for no technical reason, just to underline the fact that this class is not tightly coupled with the Ogre3D engine.

The `CustomCamera` class contains methods that allow docking of the camera behind the player's airplane, its rotation and orbiting around the airplane when in the free-look mode (mapped to the right mouse button by default) and setting its relative position with respect to the player's airplane when looking through the aiming cross-hair. It also implements a few methods for debugging purposes, such as free roaming over the landscape (these are of course disabled in the release version of the game).

4.5.8 GUID

GUID (Globally Unique Identifier) is a means of having an identifier, which will never be generated and used twice. Normally, this term refers to a 32-digit hexadecimal number, generated by a special algorithm to minimize the probability of generating [10] the same number twice. In Flying Samurai, we use a weaker 32-bit integer value. We also don't use an algorithm to generate them, but the user of the engine has to come up with a new identifier which haven't been used in the game yet.

The purpose of GUIDs in the game is to bind abstract representation of game entities (stored in `Entities.xml`) with their graphical representation (stored as records in the `Objects.xml` file). Also, GUIDs are used in the game to identify unique graphical objects (e.g. there can be more instances of the same airplane in a single mission, and while every one of them will have a different `AirplaneId` (see Section 4.4.1), all of them will have the same GUID, as they use the same mesh). Filling in the GUID value is therefore a part of the process of adding new graphical resources into the game (described in Section E).

4.6 Physics

Inseparable part of the simulation is the physics module. In short, it keeps track of the internal state of the entities (namely planes, bullets and the terrain), and updates this state every time step, taking into account user interaction (based on input translation) as well as reacts on collision events. In the following sections, we're going to elaborate further on these aspects, breaking them down and explaining the implemented code. We will focus on describing the logic that applies to airplanes as that is the core of physics complexity, but where appropriate, bullet and/or terrain behaviour is mentioned in short afterwards.

Also, as many parts of the physics can't be understood without proper theoretical background, this section also describes theory, simplifications made and implementation choices first, and only then gets to the source code citations and explanations of the basic parts.

4.6.1 Introduction

Physics module plays a big role in the project, which we can further divide into several sub-roles as follows:

1. *The core physics computing.* That basically reduces to transforming the current state indicators of the rigid body like its linear and angular velocity, position (especially elevation to compute the air density) and such and transform them into a set of forces that affect it. These will later get integrated (Bullet physics engine is used to do this uninteresting work for us) and a new object state state is computed for the next frame.
2. *Input translation.* Game's input layer provides the input events as well as their amounts which the physics module translates into change of the airplane control surfaces' positions and where necessary, recompute the parametrization of the model used in the first sub-role.
3. *Collision detection and reaction.* This is a need to go through the Bullet data structures and pick up the events that signal that a collision has occurred. For every such event it should trigger a function that analyses it and performs an action accordingly (add damage to the airplane part or remove a simulated bullet body from the simulation etc.).
4. *Logic signalling.* As it's the physics module which keeps the information about the airplane (or even pilot) damage, this information needs to be propagated. These events include when the airplane is heavily damaged (`WillDie`), pilot has crashed either to the ground or in a mid-air collision (`Dead`) or when the entity is about to be removed completely. These events are described in more detail in the game logic section (4.4).
5. *Visualization.* This role is similar to the previous one: physics module keeps track of many important values internally and there is a need to propagate them to the graphics module. This includes values like smoke level (computed from the current damage) or control surfaces' positions.

4.6.2 Model overview

First, let us state the requirements for the physics module in our project. We need to simulate the airplane physics as precisely and in as parametrizable fashion as possible. The aim is to have a full-fledged flight simulator, so especially this part is crucial. At the same time – contradictory to the previous – is the need to have a small set of simple parameters that completely describe the motion, because we also aim at the ability to conveniently add new planes to the simulation from commonly accessible historical records, without the need to perform any expensive, time consuming or other difficult measurements on a full-scale real model. Wind tunnel, to name an example, is able to produce huge amount of data that can easily describe flight

properties of the rigid body in question, but we can't design it the way that this device or such coefficients will be necessary.

After some basic research, we came to a conclusion of four basic transition functions:

1. $f_1 : v \to F$ defines the acting force given the velocity vector of the body. This can effectively cover the air friction, the lift (or portion of it, as discussed later), gravity or thrust.
2. $f_2 : \Omega \to F$, similar to the previous function, but this time it's taking an angular velocity as a parameter. Directly describes the forces that arise from a rotation of the body (a rotary engine or other bodies).
3. $f_3 : v \to M$ translates the linear velocity into a rotatory moment. This function covers the logic of turning in the direction of the air flow as well as instabilities caused by damage or rotation caused by control surfaces and alike.
4. $f_4 : \Omega \to M$ describes the transformation of actual angular velocity into a rotatory moment. The main idea for this function is to cover friction in rotation.

However, as these forces described by f_2 are often marginalised by the previous function in most applications (except special helicopter designs which we won't assume to be used in our application), we decide not to look too much further into that one. That leaves us with just three functions to consider.

Additionally leaving out the gravity (provided by the Bullet physics engine) and thrust (we'll simply add this force component to the result of the function f_1), we can approximate all these functions with a quadratic function:

$$f_i(\vec{x}) = M_i \cdot \mathrm{sgn}\vec{x} \cdot \vec{x}^2 = M_i \begin{pmatrix} x_1 \cdot |x_1| \\ x_2 \cdot |x_2| \\ \vdots \\ x_N \cdot |x_N| \end{pmatrix}$$

where M_i is some constant matrix. As we assume exactly three dimensions in our physics simulation, we can put it in the form of matrix-vector multiplication:

$$f_i(\vec{x}) = \begin{pmatrix} c_{11} & & \\ & \ddots & \\ & & c_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \cdot |x_1| \\ x_2 \cdot |x_2| \\ x_3 \cdot |x_3| \end{pmatrix}$$

The outlined model still needs some minor technical adjustments (alteration of the M_i matrix elements according to the current control surface positions or airplane damage), which we will discuss later in this text.

4.6.3 Model parameters

Having outlined the idea of the implementation of physics in the previous section, it's time to describe the set of parameters we've chosen to represent a plane. This can be further divided into parameter sets for the mounted gun, the plane body itself and finally the graphical/input interpretations. These parameters usually correspond to the ones specified in the `Entities.xml` file, with a few notable exceptions when a physical parameter is not used directly, but is still either planned in some future release or is supposed to be shown in some tables by the game logic.

To describe them in order, the list of gun/bullet parameters follows:

Accuracy Angle The higher number, the less accurate the gun is. The bullets are generated with normal distribution and this value represents the cosine of variance of the angle.

Reliability Probability that the gun will *not* jam, per bullet shot.

Range How far can the bullet fly (before it's deleted from simulation) in meters.

Rate of Fire Shots per second. The physics module doesn't interpolate this information on lower frame rates, so it might happen on some low-end hardware that the value given won't be adhered with.

Ammo The number of rounds at the beginning of the mission. Every shot decreases the counter by one, and when it reaches zero, it's not possible to shoot anymore.

Bullet Size Specifies the diameter of the bullet in meters. Even though graphics module always visualizes it as a point billboard, for physics it's a sphere with the given calibre.

Bullet Mass The weight of one bullet in kilograms.

Initial Velocity Relative velocity (to the airplane's velocity) at the moment the bullet leaves the gun barrel.

Air Friction Friction coefficient of the bullet material. Lower number will make the bullets fly a bit more straight, higher number will make it slow down rapidly after release and head to the ground.

Simultaneous Says whether all plane's gun barrels shoot simultaneously or one-by-one. The information about the barrel positions is gathered from the graphics module (refer to Section 4.5.2).

The parameters that alter the visualization or the input interpretation are:

Pitch Up Manoeuvring capabilities of the plane. 1.0 is base, 2.0 means it's twice as agile in the specified direction. It's possible to specify different values for positive and negative sense for planes are more agile in right turn but less able to do left turns for example (this is for example the case of airplanes with a rotary engine).

Pitch Down The other manoeuvring coefficient of pitch.

Roll Left Manoeuvrability when rolling left.

Roll Right Manoeuvrability when rolling right.

Yaw Left Manoeuvrability when yawing left.

Yaw Right Manoeuvrability when yawing right.

Smoke Threshold Low When computed smoke level is lower than this threshold, none will be shown.

Smoke Threshold High At the time the computed smoke level is higher than the value specified in this parameter, a maximum density will be assumed for the rendered smoke. Anything in between is linearly interpolated to fall into the $[0, 1]$ interval.

Smoke Body Coefficient that the damage caused to the body is multiplied with. Used when computing the smoke level.

Smoke Engine Multiplicative coefficient for the damage caused to the engine. Used when computing the smoke level.

Smoke General Coefficient to specify the amount of smoke for general damage (any hit to the plane is accumulated to this damage indicator).

Rudder Mul. Multiplicative constant for transforming the input value (always inside the range $[-1, 1]$) to the angle of the rudder airplane part visualization.

Rudder Add Additive constant for rudder visualization.

Elevator Mul. Multiplicative constant for elevator visualization.

Elevator Add Additive constant for elevator visualization.

Left Aileron Mul. Multiplicative constant for left aileron visualization.

Left Aileron Add Additive constant for left aileron visualization.

Right Aileron Mul. Multiplicative constant for right aileron visualization.

Right Aileron Add Additive constant for right aileron visualization.

And finally, the parameters mainly used in the core model:

Engine Power Engine power, in Watts.

Empty Mass Weight of the plane fully unloaded. This number is assumed to be used in airplane listings, as physics module ignores it and uses the next one instead.

Base Mass Weight of the plane with pilot and other necessities (i.e., the base mass of the body that is simulated).

Fuel Mass Weight of the full fuel tank.

Endurance Time (in hours) how long the fuel lasts. Used to compute the mass change during the flight or the speed of the fuel depletion.

Range Range (in kilometres) how far the plane can fly. It's an informative value, not used in the simulation.

Maximum Speed Maximum forward speed (to reach at sea level-like air density at maximum thrust). This value only specifies the forward friction (engine power is given) and is not actually used to cap the speed (one can achieve higher values by utilizing gravity). Specified in meters per second.

Take-off Speed Minimum speed required to achieve balance of gravity and lift (any faster and the lift will exceed the gravity, resulting in takeoff).

Stall Speed Another value not directly used in physics simulation and intended to be shown to user for simple a comparison purposes.

Service Ceiling Service ceiling specifies the altitude in meters where the air density decline causes the rate of climb to decrease to zero.

Rate of Climb Rate of climb is specified in m/s at the sea level at full thrust. It represents the ability of the plane to counter the gravity with its lift. As stall speed and take-off speed can be (to a small degree of error) derived from this value (it basically gives us lift), those parameters are not actually used in simulation (assumed to be shown in plane listings though).

Wing Area Wing area is used to recompute the lift force (the one derived from the rate of climb) in situations like wing damage. Specifies the cumulative surface of all wings in square meters.

Wing Alpha α or an angle of attack of the flow of the air to the wing. The correspondence with the lift is not that clear (it's only one of the major forces, together with low pressure above the wing at high speeds).

Wing Loading This value specifies the amount of weight one wing can support in kilograms. Could be used in wing bending visualization, but it's not implemented at the time of writing this document.

Surface X Describes the surface of the projection of the three-dimensional model along the X axis (that is, Y-Z plane). Together with the following values it's useful when computing the friction coefficients along the sides.

Surface Y Surface in square meters of the model projection along the Y axis (X-Z plane).

Surface Z Surface in square meters of the model projection along the Z axis (X-Y plane).

Rotational Friction X One of the less straight-forward "plane agility" parameters. This friction specifies how "heavy" the plane body feels when turning – low value means it's very likely to turn around the given X axis, higher value represents relative stability when turning.

Rotational Friction Y Friction in rotation along the Y axis (from the top to bottom).

Rotational Friction Z Friction in rotation along the Z axis (one aligned with the direction of flight).

Rotation Correction 1 This correction is an empirical value that specifies how likely the plane is to turn into the velocity vector direction. When one uses large values, the plane body will be very likely to turn into the direction of the air flow and relatively difficult to effectively steer. The first value represents the $Y \to X$ direction.

Rotation Correction 2 The other correction coefficient, this time representing the $X \to Y$ sense.

Length Dimension of the plane body. The only use is in plane listings, as this value is represented much better in the plane mesh.

Wing Span Width of the plane.

Height Height of the plane.

Minor Impulse Lower limit for applied impulse (collision reaction force – when the plane hits other plane or a ground). If the actual applied impulse is lower than this one, it's ignored.

Major Impulse High threshold for applied impulse. This extreme will signal instant death.

Material Friction The material constants of the plane. As planes rarely interact with the ground or other planes (and assumption goes that it doesn't play an important role for bullets), we simplify the physics by using only one material for the whole plane body.

Material Restitution Restitution coefficient of the rigid body representing airplane.

Damage Lift Degradation Specifies a coefficient (in $[0, 1]$) how the lift of the airplane is altered by the damage.

Damage Stability X How the stability of the flight (right-left axis) is altered by the damage. Range $[-1, 1]$, positive numbers are accepted and mean it'll be harder to rotate along the given axis.

Damage Stability Y Damage-stability dependence coefficient for top-bottom axis.

Damage Stability Z Damage-stability dependence coefficient for forward axis.

Damage Torque Coefficient for damage-torque dependence.

Damage Wing Imbalance How strong will the effect of difference in wing damage on the airplane's imbalance be.

Zero Lift-Drag Coefficient Unused tabular constant.

Lift-To-Drag Ratio Unused tabular constant.

Rudder Area Unused tabular constant.

Aileron Area Unused tabular constant.

Elevator Area Unused tabular constant.

All these parameters are specified in the `Entities.xml` file, using camel casing (plus few minor non-physics related ones).

4.6.4 Core formulae

Let's break up now, one by one, the formulae used in the game, starting from the basic ones building our way to finally express the force and the torque that affect the plane rigid body. This way, we're going to achieve a similar dependency and order of computation that is actually used in the source code (some excerpts will come in later section).

All the following are coded in `PlanePhysicsModel` class, which is used (and can be shared) by every plane that uses the same set of model parameters. It possesses the ability to compute (even when constant) most of the physics.

Air density Given the altitude of the plane (h) and the density scaling constant of $\rho_0 = 1.2\frac{\mathrm{kg}}{\mathrm{m}^3}$ (assuming temperature of 20°C), $g = 9.8\frac{\mathrm{m}}{\mathrm{s}^2}$, $p_0 = 101\,325\mathrm{Pa}$, we compute

$$\rho(h) = \rho_0 \cdot e^{\frac{-h\rho_0 g}{p_0}}$$

Thrust Given a constant angular velocity of the propeller, the device is providing different thrust force depending on the surrounding fluid flow. As we aim for realism, we need to describe this phenomenon. Given the power of the engine (parameter *Engine power* or P in the following formula) and a current absolute length of the velocity vector ($l = |\vec{v}|$), we use the relation

$$F_t(\vec{v}) = \begin{cases} P & l \leq 1 \\ \frac{P}{l} & l > 1 \end{cases}$$

The first option is not physically accurate, but we use it as a workaround in our discrete model. It's actually true that we can assume the force acting on a rigid body is infinitely large while the object is not in motion. However, as soon as the object gains some speed, this force very sharply decreases and thus, integrating over a continuous function describing this behaviour would still result in a finite number. We have no such privilege and so we assume the velocity is never below $1\frac{\mathrm{m}}{\mathrm{s}}$.

Also, in the formula we should use only the portion of velocity parallel to the direction of flight, but we have made a second assumption that most of the time the body will be in such a situation anyway.

Both these simplifications lower the actual power of the engine during the time airplane is contained in one of these special circumstances, so one might wish to set a slightly higher *Engine power* parameter if necessary.

Physical matrix set As stated earlier, the core of the physics is driven by transformation of the current linear (angular) velocity to force or momentum. This is achieved by the multiplication of the altered vector (where every value is squared and multiplied by the signum of the original value). Details were already introduced in the description of the functions $f_i, i \in \{1, 2, 3, 4\}$.

Here, we're presenting the formulae for the (eight) matrices that are necessary to define the above mentioned functions. The indexes are correlated with the functions, a positive matrix is called M_i^+ and a negative one M_i^-.

$$
\begin{aligned}
M_1^+ = M_1^- &= \begin{pmatrix} -c_{fx} & 0 & 0 \\ 0 & -c_{fy} & lift \\ 0 & 0 & -c_{fz} \end{pmatrix} \\
M_2^+ = M_2^- &= \begin{pmatrix} 0 & c_{r1} & 0 \\ -c_{r2} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
M_3^+ = M_3^- &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
M_4^+ = M_4^- &= \begin{pmatrix} -c_{rfx} & 0 & 0 \\ 0 & -c_{rfy} & 0 \\ 0 & 0 & -c_{rfz} \end{pmatrix}
\end{aligned}
$$

where c_{r1} and c_{r2} are rotation corrections and c_{rfx} to c_{rfz} are frictions in rotation. These parameters are directly specified by the designer of the model and read from the XML file. c_{fx}, c_{fy} and c_{fz} are friction coefficients and $lift$ is naturally a lift coefficient. These are computed as follows

$$
\begin{aligned}
lift &= -\frac{(m_{base} + m_{fuel})g}{v_{term}^2 \rho(h_{ceil})} \\
c_{fz} &= \frac{F_t(v_{term})}{v_{term}^2 \rho(0)} \\
c_{fy} &= -lift \cdot \frac{v_{term}^2}{v_{climb}^2} = -lift \cdot \left(\frac{v_{term}}{v_{climb}}\right)^2 \\
c_{fx} &= \frac{S_x}{2S_z} c_{fz} + \frac{S_x}{2S_y} c_{fy}
\end{aligned}
$$

where the parameters are

m_{base}	base mass
m_{fuel}	fuel mass
v_{term}	terminal velocity in forward sense
h_{ceil}	service ceiling
v_{climb}	climb rate

As we see, matrices are basically zero in every element where we didn't mention otherwise in the introduction or where our God Juju on the mountain didn't tell us. Also, we mention (and actually use in our code) the M_3 matrices and compute f_3 ($\omega \to F$), but leave it zero. This is for easy future extension for new models – in such a case, it would be easy to reuse, alter or overload (with slight improvements) the existing class.

Another noteworthy thing is that although we compute exactly the friction in the y and z-axes, the parameters don't contain enough information to express the similar coefficient in the x (right-left) axis. To make up for this insufficiency, we interpolate it from the previous two. This should not make a considerable difference, as this friction coefficient plays a marginal role in the big picture.

Bullet, on the other hand, follows an extremely simple set of formulae. It's basically shot at its initial speed (which gets summed with the current velocity of the shooting airplane) at the destined exit point (provided by the graphics module). Then, the gravity is the only force that acts upon the bullet. When the bullet hits another body, it is removed from the simulation, so there are no more interesting aspects about this collision object.

Equilibrium velocity When initializing an airplane, one needs to set such an initial speed that the plane will not receive a significant kick (either from friction or by the pull from the sudden thrust). This is achieved by a method in `PlanePhysicsModel` class that computes the following approximation.

$$\begin{aligned} f &= f_1(\vec{v}) \\ v_e &= \sqrt[3]{\frac{t}{|f|}} \end{aligned}$$

f_1 is defined in the same sense as we have introduced in the Section 4.6.2.

4.6.5 Input processing

Physics module can get all the input for every player as a linked list. It iterates through this list and performs every event accordingly. For all the main inputs there are two kinds of them: *set* and *relative change*. While the first one changes the internal value right away, the relative change is pre-processed, added to the current stored state and finally capped to still fall into the necessary interval (usually $[-1, 1]$).

Pitch, roll and yaw Every one of the three axis (representing pitch, roll and yaw) has a *left* (*top*) and *right* (*bottom*) coefficients. Whenever an input is read that is in relation with the given axis, the difference is multiplied by the appropriate coefficient (depending whether the input is in the positive or negative sense) and clamped back to the interval $[-1, 1]$. This is kept and used in the core formulae.

Thrust Thrust is stored either as a percentage of the full engine output (an integer number in the range $[0, 100]$) or as a floating point number from $[0, 1]$. These representations are used interchangeably on a few spots in the code and the first one is the native representation for the input chain. Any change to the thrust (either relative or absolute setting) is applied immediately (we assume the rotary momentum of the engine is marginalized by its output strength).

4.6.6 Visualization

As we've tried to simplify the graphics module to perform as little of non-graphic related tasks, the decision was made to move part of visualization to the physics module. The aspects in question are the smoke level computations (as the damage model is pretty much an internal part of physics) and control surface deviation. The later is not as much an obvious choice, but is still bearable as physics module translates input and this way it can be expanded to compute the fluid body dynamics more precisely.

Smoke level computation is based on three internal values (ranging from 0 to 1) defining the *body* (D_b), *engine* (D_e) and *general* (D_g) damage. These are combined in a specified manner using the loaded parameters *Smoke Threshold Low* (t^-), *Smoke Threshold High* (t^+) fulfilling the inequality of $t^- < t^+$,

Smoke Body (s_b), *Smoke Engine* (s_e), *Smoke General* (s_g) using the formulae:

$$\begin{aligned} \Sigma s &= D_b s_b + D_e s_e + D_g s_g \\ s' &= \frac{\Sigma s - t^-}{t^+ - t^-} \\ s &= \begin{cases} 0 & \Sigma s \leq t^- \\ s' & \Sigma s \in (t^-, t^+) \\ 1 & \Sigma s \geq t^+ \end{cases} \end{aligned}$$

$s \in [0, 1]$ is sent to graphics module as an amount of smoke to render coming out of the particular airplane entity. It's interpreted as an opacity of the smoke.

Control surface angles is the other derived value set managed by the physics module. The physics is the thread to process the input, it keeps the input translation constants and also returns the rotation of the control surfaces (relative to the position set in the mesh – see Section 4.5.2).

The graphics module will need an angle in radians (representing the rotation along the axis from the plane model). To get this value, all of the three control surface groups have additive and multiplicative constants. Given that a is the additive and m the multiplicative one and $i \in [-1, 1]$ the stored input, returned value v will be equal to the obvious

$$v = m \cdot i + a.$$

4.6.7 Damage model

Airplane body division An airplane mesh is divided into sub-meshes, where every one plays a special role, in graphics as well as in our damage model. For physics purposes, we divide it into

1. body,
2. elevator,
3. engine block,
4. rotary engine,
5. pilot cage,
6. propeller,
7. rudder,
8. left and right wheel,
9. left and right wing.

For each airplane part, a mesh is requested from the graphics module. If Juju gives us the power, the game will not collapse into the threading hell. These parts are added in order into one `btCollisionObject` abstract class inherited object, with user pointer set to the `CollisionData` instance holding information about the part. This pointer will be used when a collision occurs to identify the behaviour branch in the decision tree.

Let us also note that to make the movement possible, we need to convert the mesh into a convex hull, as the general mesh is unable to represent a non-static (that is, with a zero mass) object in Bullet collision

detection and reaction library routines. Supposedly this feature is there to speed up the already pretty slow computation pipeline and to solve some extreme situations occurring in the discrete simulation resulting in the artefacts in the object movements. One such example could be a quickly rotating concave object.

Damage variables To describe a damage caused to either part of the airplane body, we're going to keep four variables (ranging in $[0, 1]$). Every collision with another object will alter one of these variables. Also the game engine will keep a set of flags (boolean vales) that indicate some of the special cases (like whether the engine is not operational or a pilot is dead). The variables are namely

1. engine damage d_e,
2. general damage d_g,
3. left and right wing damage, in order d_l and d_r.

How these are altered can be deduced from the table 1

Model alterations These variables, together with the other values specified in the parameter set will be used to compute the *damage matrices*, which are added to the already computed M_i matrices defining functions f_i to alter the physics model. For example, to decrease lift, add some permanent torque (resulting from the wing damage imbalance) and similar.

Actually, only the values d_g, d_l and d_r directly alter the physical model. The value of d_e (engine damage) only adds up over the game period and if on any occasion happens to reach 1, the engine is stopped and it's not possible to start it again (the physics module will make sure the thrust percentage will be set to 0 until the end of the mission). All the other controls are still in effect, just the plane is bound to land sooner or later.

As for the other three parameters, they are used to decrease lift (to make it harder to keep the plane in the air), increasing friction in the forward direction (thus making it much harder to glide), adding to the rotation correcting coefficients in M_2 (so the player will find it harder to steer to the desired orientation) or just to add a permanent torque to turn the plane along the forward axis (because one of the wings has got a greater lift than the other). Let's describe the formulae used to simulate these aspects.

Lift and stability reduction As these forces are described by the same matrix (the one that defines $v \to F$ transition), one can unify these aspects. The additive matrix (to either M_1^+ or M_1^-) is

$$\begin{pmatrix} ds_x \cdot \frac{d_l+d_r+d_g}{3} c_{fx} & 0 & 0 \\ 0 & ds_y \cdot \frac{d_l+d_r+d_g}{3} c_{fy} & -dlg \cdot \frac{d_l+d_r+d_g}{3} \cdot lift \\ 0 & 0 & ds_z \cdot \frac{d_l+d_r+d_g}{3} c_{fz} \end{pmatrix}$$

where ds_x, ds_y and ds_z are *damage stability X, Y, Z* parameters and dlg is *damage lift degradation* parameter. $lift$ is the value as computed in Section 4.6.4.

Adjustment of rotation correction and torque Again, these two aspects of damage can be combined into one matrix (the one that alters the positive or the negative one defining the $v \to M$ function). The matrix is

$$\begin{pmatrix} 0 & tor \cdot \frac{d_l+d_r}{2} c_{r1} & 0 \\ -tor \cdot \frac{d_l+d_r}{2} c_{r2} & 0 & 0 \\ 0 & 0 & wi \cdot ws \cdot (d_r - d_l) \end{pmatrix}$$

where tor is *damage torque* parameter, wi is *damage wing imbalance* and ws the *wing surface* parameter. c_{r1} and c_{r2} have the same meaning as in 4.6.4.

Table 1: Hard-coded coefficients for damage per every hit to the given airplane part

From the following table one can read the amounts added to the each damage component when a given airplane part is hit. There is a special behaviour when hitting a pilot cage (no damage added, but the pilot status is set to dead, blocking any successive input from the player/AI module. Also, although right and left wheels are defined and there is a switch for them on many places in the code, landing have never been finished, so it's basically an empty instruction.).

Plane part	left wing	right wing	engine	general
Body (left part)	0.05	0.00	0.00	0.02
Body (right part)	0.00	0.05	0.00	0.02
Engine	0.00	0.00	0.40	0.10
Pilot cage[a]	0.00	0.00	0.00	0.00
Left wing	0.05	0.00	0.00	0.00
Right wing	0.00	0.05	0.00	0.00
Left wheel	0.00	0.00	0.00	0.00
Right wheel	0.00	0.00	0.00	0.00
Rudder	0.00	0.00	0.00	0.10
Elevator	0.00	0.00	0.00	0.10
Propeller	0.00	0.00	1.00	0.00

[a] causes instant death for pilot

4.6.8 Collision reaction

To provide a reliable collision against a potentially infinite heightmap, the game engine introduces a relatively small piece of heightmap copy under every plane. The shape and the position of this heightmap is updated every swap in physics module's `SwapHook` method.

During a single simulation step, all collisions are collected into a linked list and stored. In the last phase of the physics thread computation, this list is retrieved and every item in it is processed. Basically we separate four different situations.

1. airplane-airplane collision,
2. airplane-ground collision,
3. airplane-bullet collision and
4. any other collision (basically this leaves out with bullet-bullet or bullet-ground).

Airplane-airplane collisions For every such a collision, the applied impulse (for both planes involved) is computed (which is handled by the Bullet library) and compared to the *impulse minor* and *impulse major* parameters. A corresponding amount of damage is added to the d_g damage indicator and if it reaches 1, the plane is considered dead.

Airplane-ground collisions For this case we compute basically the same impulse logic as in the airplane-airplane collisions. However, in this case the Bullet library is also utilized to put the collision detection to rest when the body settle in some stable position. We use a technique known as deactivation available in the library (it needs to set some limits, which we've hard-coded). To activate the plane, the function `PlanePhysics::CalcPhysics` checks whether the forces applied to the plane body are of any significance.

Airplane-bullet collisions When such a collision occurs, the bullet is naturally removed from the simulation, but just before that, the shooting pilot Id gets recorded for mission logic purposes. Also, according to Table 1, a given amount of damage is dealt (or a flag is set) to the plane. The damage model as described in the previous section will take care of the rest.

Other cases In other cases, the game engine just simply removes any bullet involved. That is, if two bullets collide, they will be removed from the simulation. Also in case a bullet hits the ground, it will get removed.

4.7 AI

The objective of the artificial intelligence used in the game is to perform as fairly as possible. The idea originated from the materials that pointed out that during the World War I., it was a sneak calm strategy more like than any other that lead to victory. Fighters that were able to gain an advantage over the enemy (be it the altitude or the position in their back) won the dogfight. So we strived to provide a modern player with a very similar realistic experience. Things like hiding in the sun or planning a path through clouds and other ways to block the clear vision were discussed, however, never completed.

AI design comes mainly from the early stages of the development, that means, it contains plenty hooks and abstract classes that were targeted to give the computer a controlled plane to provide such a high-level decision-making options. However, as many of the originally intended aspects were either not implemented at all or there was no consensual feasible solution implemented to transfer the internally stored structures either to physics or AI module, the abstract classes and drafts are all that remained.

4.7.1 Overview

The developed artificial intelligence system consists of another three significant (in both complexity and size) subsystems. For a better orientation for the reader, we're going to very quickly introduce these. Later, there is an entire section dedicated to each one of them, so the reader is free to decide which one to read.

Used libraries Most of the software used in the AI module comes from our team – including the neural network recall and training framework. From third-party software, we used the middle-level OpenSteer steering and geometry library [20]. Physics (necessary to rank the inverse motion decisions) is being computed by our physics module described earlier in this document (in Section 4.6). It uses Bullet library, although for performance reasons, this library is used only indirectly (for purpose of the calculations in the AI we disable almost all computations like collision detections and similar). Full-scale computations are performed only after the AI makes a final decision and sends it via the reader-writer subsystem as an input equivalent to the input from any other human player.

As the knowledge of these libraries (mainly of OpenSteer) will be required in the following text extensively, it's recommended for the reader to understand at least the main features and usages of these libraries.

Information gathering Computer-controlled player receives imperfect information about the surrounding environment mainly via two classes: `Radar` (returns the list of other surrounding agents – friendly as well as enemy ones) and `HeightMap`, which provides an abstract interface returning a set of nearby objects to avoid. Especially note the very vague definition of the latter: even though the name suggests it ought to represent just a wavy terrain, it can do much more. As it returns a list of objects (triangles, spheres, etc.) to avoid, it can also represent terrain features like trees, houses, even cavities.

This information is gathered and given to the agent to process.

Agent An agent is represented by the unifying class `PlaneAgent`, which gathers the decision information and either uses it directly (when there is an object to avoid in an immediate distance) or uses it as a parameter for the *agent role*. It is also a successor of the `OpenSteerPlugIn` class, so it can be easily visualised and debugged in the OpenSteer infrastructure. It takes the behaviour prescribed by its agent role defined for any situation the plane can get into and returns one of the (later described in more detail) states, like "catching another plane", "evading from enemy fleet" or similar.

Every such a state (and possibly one or more associated parameters) can give an instructions what is the plane supposed to do. It is also translated in its geometric meaning (considering the current plane speed, position and orientation as well as any limits given). This is usually a curve that the plane should follow to complete the task. For our purposes, we only get a point few milliseconds (two physics simulation frames to be exact) in the future and try to adjust the control surfaces of the plane to reach that point. To get this done, we're introducing another AI subsystem, namely the *inverse physics*.

Inverse physics Inverse physics or as a similar functionality is usually called in robotics – the inverse motion – is a way to translate the current and future positions in the world into a set of elementary instructions that the agent should perform, ideally resulting in reaching the later position. In our case, the elementary instructions represent the control surface alterations (like moving the left aileron upwards) or engine thrust change.

Because we target as fair AI as possible, we can't settle on skipping this step like many other arcade games do. A good simulator needs to make sure every player, including the computer-controlled one, is adhering to the laws of physics, at least the ones that are defined in the simulation. This system will make sure the AI can't fly higher, faster or slower, without losing or gaining altitude accordingly, which is exactly what we need. On the other hand, the inverse motion system often suffers from high dimensionality. The set of input parameters (at least two position vectors and two rotation quaternions, all floating point numbers with vaguely defined extremes) makes it a tricky theoretical computer science problem to solve.

The description of the three main subsystems (and how they propagate the values into physics and later read the simulated results back) can be seen on Figure 16.

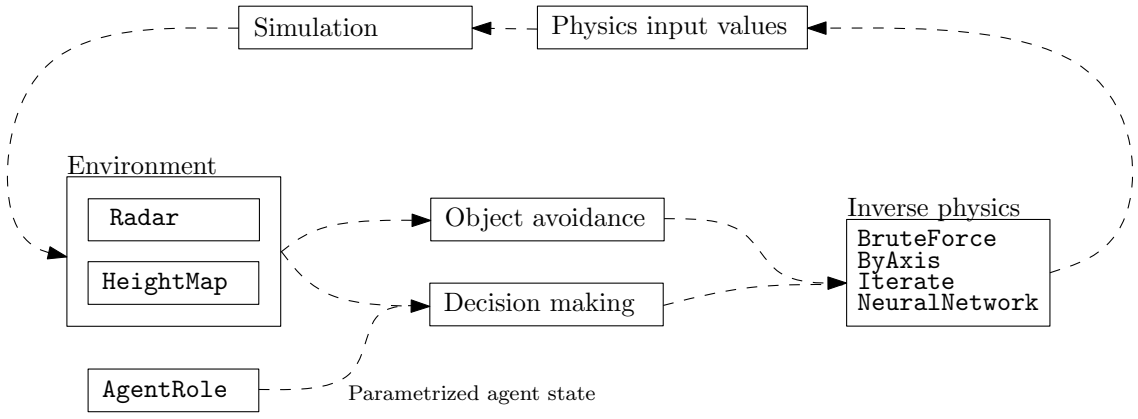


Figure 16: A scheme of value propagation inside the AI subsystems as well as the physics module.

4.7.2 Environment knowledge

Here is an overview of the implementation details. The base abstract class in question will be described (with a code listing attached) and then the actual implementation will be explained in short.

Radar As can be observed from Figure 17, the `AbstractRadar` class holds the lists of agents (separated into a list of friendly and a list of enemy planes) in the given radii. The *Discover* radius specifies the minimum distance when the radar can consider informing the AI about the presence of another plane. As soon as a plane is discovered, a human player somehow keeps an eye on him, so it might be harder to lose

track about him. To emulate this, there is a (in general case) different value for *lost radius*, which makes sure the radar will not consider removing the agent from the sets of the known ones before it's at least the given distance from the computer player.

```
class AbstractRadar
{
public:
        void SetMaxRadius(float _discover, float _lost);

        virtual void Update(float _dt, PlaneAgent* _agent) = 0;

        void GetNeighbors(PlaneAgentGroup &_neighbors);
        void GetNeighborsBySide(int _side, PlaneAgentGroup &_friends,
            PlaneAgentGroup &_enemies);
        PlaneAgent* GetLostNeighbor();

        void RemoveReferences(const PlaneAgent *_agent);
        void Reset();

        AbstractRadar();
};
```

Figure 17: Public `AbstractRadar` methods

An update is always performed inside the `Update` call, which takes the time elapsed (called dt) and an agent instance in question. After such call, one can fetch the updated lists, either using one of the `GetNeighbors` functions (for visible agents) or to fetch one-by-one the list of neighbours that has been lost.

`RemoveReferences` is a technical method that ensures there's no reference to a specified agent left in the radar. The sole purpose is to make sure the engine will not crash on an invalid pointer operation once an agent is freed from memory.

In the final stage, we're using an *angular exponential radar* implementation in the `AngularExponentialRadar` class. It takes three parameter pairs (multiplicative m and additive a) plus two angles (for front and rear). The angles are used to detect whether another plane is in front (resp. rear) of the airplane centre, that is, whether it is in the cone around the `forward()` (resp. `-forward()`) vector. The two areas are visualized on Figure 18. Anything outside these two cones is considered as "side".

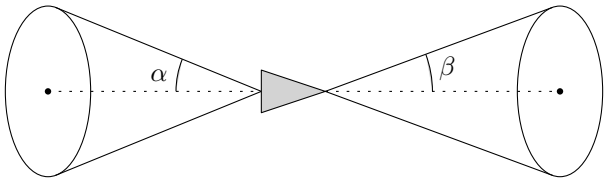


Figure 18: The front and the rear area of the airplane parametrized by two angles in an angular-exponential radar implementation.

Once we know the relative position of the agent in question, we can compute the probability of seeing it by using the other two parameters and the formula

$$p = e^{-mx-a}$$

where x is the distance. This probability is recomputed every second, simulating a pilot that takes a good look around every now and then. It also (when parameters are chosen well) can make a nice difference whether the player is approaching the AI from the front or rear, giving the player a nice realistic feeling that the AI still haven't seen him.

Height map Height map, on the other hand, has a comparatively simple interface. As can be seen in Figure 19, there are just two virtual methods. The first one, `AddObstacles`, pushes new obstacles to the 'given by reference' obstacle group for a given vehicle and radius. OpenSteer logic will be later used to avoid these obstacles. This significant level of abstraction allows us to add any number of OpenSteer or OpenSteer-derived obstacles into the list, like cubes (for houses), planes (large levelled terrains, bodies of water etc.), or triangles (for an exact height map).

`GetHeight` is even simpler method, which just returns the buffer below an agent. The main purpose of this function is to signal to the higher level logic (namely the `PlaneAgent` class described in just a moment later) whether it's appropriate to switch into the "gain altitude, you're too low" state.

```
class AbstractHeightMap
{
public:
        virtual void AddObstacles(OpenSteer::ObstacleGroup &_obstacles, const
            PlaneAgent *_vehicle, float _radius) = 0;
        virtual float GetHeight(const PlaneAgent *_vehicle) = 0;
};
```

Figure 19: Public `AbstractHeightMap` methods

The concrete implementation is rather simple, it just takes the highest point on the terrain in the given radius and returns a single plane obstacle with this height. This solution is very beneficial considering its complexity (OpenSteer will get either no or just one obstacle to avoid). Also, the original plans were not found viable as there is no reliable method to query objects from the graphics module (just a function for querying the height at a given point).

4.7.3 Agent states

In preparation for the next section, we're going to describe each agent state and possibly its parametrization here. In most cases they simply copy the OpenSteer manoeuvre naming conventions.

Heading This represents a need of the agent to reach a single point in space. For example, when an agent needs to return to the base, assuming he possesses the coordinates of the base, it can signal it is *heading* towards those coordinates.

Approaching Almost the same like heading, just with an extra parameter specifying the time. The agent needs to *approach* a given point in space in a given time. The speed will be adjusted accordingly. However, this manoeuvre (unlike the later described catching) doesn't specify anything about the orientation itself (thus is not really usable for landing, for example). On the other hand, it's perfectly fit in cases an agent needs to reach a known point on the front in, say, ten minutes.

Following Similar to heading, just in this case an agent is *following* a *path* at an *offset*. Path also has a specified radius, in which an agent can operate more or less freely (however, the position on the path plus the offset will result in an attractor). If the radar reports some other friendly vehicles, it will use their positions to keep near the nearest point on the path to the average position of the group. In combination with a skilful setting of the offsets will naturally result in the set of agents who keep a formation. On top of that, if the *strict following* parameter is not set, the agent will slow down (or even turn around) if it detects some other plane(s) in the group lag behind him as well as will try to catch up in case he's behind.

Taking into account the game purpose, the agents will most probably spend a significant time in this state. Either when escorting a bombardiers (when the path is set the same for all agents, bombers follow strictly and fighters will switch to catching whenever an enemy is spotted) or patrolling the front (where the setting is obvious), following logic plays a central role.

Evading Evading is a manoeuvre parametrized by a list of other agents. These will be considered repellent and the steering will result in *evading* all these agents.

Catching The most sophisticated state is no doubt the *catching* another plane (called *prey*). It needs to get an advantageous position over the enemy (that is, get behind and possibly slightly above) as well as it needs to adjust the orientation accordingly to get into the shooting position. While doing all this, it also dictates to predict the prey's future position (to decide when exactly to shoot and where to steer to).

It is parametrized by the *far* and *near* distances, where the first one signifies the distance that is too far to shoot, while the later is the time to slow down or steer away from the prey (as it is dangerously close and a collision might be imminent).

Gaining height The one manoeuvre that translates into another OpenSteer logic. It basically means the plane nose should level with the ground, keep stable straight flight and gain heigh (not too quickly though, respecting the airplane possibilities). It's not triggered by the agent role itself, but by the `PlaneAgent`.

Wandering This is not really a state on its own, but it is a parameter for any of the previous states. It says how much an agent might want to *wander* off the otherwise computed path. It allows to simulate more human-like behaviour, because it will on occasions (randomly) steer few metres to the side, higher or lower. Might prevent lock-ups, when for example two agents decide to catch each other and both possess the same plane. Such a situation (without this random "wandering" around) would result in two planes in a tie – circling, trying to get into the back of each other.

4.7.4 Agent role

Agent role describes what an agent (computer-controlled plane) should chose to do (namely the agent state as described in Section 4.7.3). Similarly as in the case of the radar or height map, we've designed an abstract class for simple design and introduction of the new strategies.

The first four have informational role for other computer-controlled players and roughly mean the following

Revenging is an airplane when it will shoot at me after I open fire (true for most players but bombers and such).

Defensive is such player who when entering area he controls, will open fire (agents who protect a base, for example).

```
class AbstractAgentRole
{
public:
        virtual bool IsRevenging() const = 0;
        virtual bool IsDefensive() const = 0;
        virtual bool IsOffensive() const = 0;
        virtual bool IsWandering() const = 0;

        virtual void SituationUpdate(const std::vector<PlaneAgent*> &_friends,
            const std::vector<PlaneAgent*> _enemies) = 0;
        virtual void AgentDisappear(const PlaneAgent *_agent) = 0;

        virtual AgentState GetState() const = 0;
        virtual bool Shooting() const = 0;

        virtual std::vector<PlaneAgent*> GetEvadeList() const = 0;
        virtual OpenSteer::Vec3 GetHeadingPoint() const = 0;
        virtual OpenSteer::Pathway* GetPath() const = 0;
        virtual OpenSteer::Vec3 GetPathOffset() const = 0;
        virtual bool GetPathStrictFollowing() const = 0;
        virtual float GetApproachTime() const = 0;
        virtual PlaneAgent* GetPrey() const = 0;
        virtual void GetPreyRanges(float &_close, float &_far) const = 0;
        virtual float GetWanderWeight() const = 0;
};
```

Figure 20: Interface of the `AbstractAgentRole`

Offensive describes a player who actively searches for enemies and shoots them down.

Wandering is a flag for vehicles that move around without a mission (not related to wandering agent state). Opposite is a plane that follows a path or circles above an important point (guards the air territory).

The decision making of other agents heavily depends on correct return values of these four functions (as it might make a difference of attacking or fleeing).

`SituationUpdate` and `AgentDisappear` (meaning disappearance either because the agent have died or flew away) are functions that update the internal state of the agent. The derived class should utilize these to compute what should an agent do given the information about other players.

The numerous (nine) getters at the bottom of the interface definitions are just to get the internal agent state and its parameters, as described in the dedicated Section 4.7.3.

Simple agent Simple agent's behaviour is the base for the other two. This player will take a path and other parameters when being created and during the gameplay will basically switch between *following* (or just plain wandering if not path is specified), *catching* (when sees some enemy) or *evading*, when enemies outnumber the friendly planes the agent knows about. It chooses as prey any (the first seen) enemy.

Fighter Fighter is derived from the simple agent and is all defensive, offensive and revenging. It also follows more sophisticated logic when choosing a prey, when it prefers undamaged (that is, ignores wounded

players which usually only wait for gravity to crash them) planes and among multiple undamaged possible targets chooses the offensive ones. As a real flying Samurai, never evades anyone and rushes into the fight.

Bomber Bomber role, although derived from the simple agent, is way simpler. It basically never changes its state from the initial following the path. It also never returns the fire, is not defensive nor aggressive. Bombers usually didn't have any mounted guns anyway.

4.7.5 Decision making

Decision making is stored in the `PlaneAgent` (significantly extending the base class from OpenSteer) `PlaneAI` (consisting mainly of a routines to get a set of decision values from the former) class. Let's take a look at most important methods and members in the Figure 21.

```
class PlaneAI
{
private:
        InverseGuess *m_inverse;
        PlaneAgent *m_agent;
        AbstractHeightMap *m_map;
        SimpleAgentRole *m_role;

public:
        PlaneAI(ProximityDatabase &_pd, AbstractHeightMap *_map,
            SimpleAgentRole *_role);

        void SetParams(const Entities::AirplaneType &_params);
        void SetPathway(const std::vector<Ogre::Vector3> &_points, const
            Ogre::Vector3 &_offset, float _radius);
        void SetSide(int _side);
        void SetWounded(bool _wounded);

        void SetPosition(const btTransform &_transform, btScalar _velocity);
        void ComputePosition(float _time, float _elapsed);
        btTransform GetTransform() const;
};
```

Figure 21: Extract of some important definitions in the `PlaneAI` definition

One can observe many are named quite intuitively. The class holds record of the actual agent in the form of the `PlaneAgent` pointer as well as some access methods. The core usage is to get the current position (represented by a Bullet `btTransform` structure containing both position and orientation plus a velocity vector), let the plane agent decide the future position (by a call to `ComputePosition`) and then get the target transform that will be later handed to the inverse physics subsystem.

Another class we mentioned earlier is the `PlaneAgent`. This is much bigger piece of code, so again let's list the main defined features and describe them in more detail (figure 22).

One can observe the multiple levels on which this class needs to function. It was very favourable especially in the beginning stages of the development (when the game was not playable yet) to use the class purely as an agent being visualized in the OpenSteer program. That's why we needed to implement `draw()` method (together with some handy annotation functions not shown in the listing). Also, `reset()` and `update()`

```
class PlaneAgent : public OpenSteer::SimpleVehicle
{
public:
        PlaneAgent(ProximityDatabase& _pd, AbstractHeightMap *_heightMap,
            AbstractAgentRole *_role);

        void draw();
        void reset();
        void update(const float _currentTime, const float _elapsedTime);

        // ...

        void GetNeighbors(float _radius, PlaneAgentGroup &_neighbors);
        OpenSteer::Vec3 DetermineCombinedSteering(const float elapsedTime);

        friend class PlaneAgentPlugIn;
};
```

Figure 22: Most important parts of the `PlaneAI` definition (*not* the complete code listing)

methods are abstract functions defined in `AbstractVehicle` and play a role when this class is used in OpenSteer, but the AI module (or, to be more precise, plane AI class) will utilize them for our purposes of decision making in the game.

This class does the geometry itself, according to the state returned from the agent role. It also adds two important aspects of the logic: it switches to a state `sGainHeight` for gaining height when the plane is very low above the ground and decides whether it's in the good position to shoot, relative to the prey. The later is always logically added to the return value from the agent role. The reason to portion out this decision whether to shoot is that an agent role has very limited knowledge about the geometry of relative prey's position. The idea is to leave as much geometry computations to the OpenSteer (and thus, `PlaneAgent`) as possible, while keep agent role class a high-level one.

We can also observe other technicalities, like the simple fact that `PlaneAgent` is the code which keeps the height map information, the obstacle of the agent itself or the fighting side. It also counts and keeps track of the friendly and enemy vehicles spotted on the radar. All this information is then served to the agent role and the decision is used to compute the geometry of the trajectory.

4.7.6 Inverse motion

Brute force This method just searches in the given range and given step size which combination of inputs gets the closest to the desired result. So, its complexity is $O(pn^4)$, where $O(p)$ is the complexity to compute physics score guess and n represents an amount of steps to try in each axis. If we assume the question of the inverse motion system in every step is simply "Do I want to increase or decrease the thrust/pitch/roll/yaw by a given fixed step?", the n will be equal to reasonable 3 (selecting from the set $-s, 0, s$). Nevertheless, this method performs very poorly when it comes to computational time.

Brute force per-axis Very similar to the previous brute force method, but this time utilizing a simple trick. We find the best scoring change in every axis separately and then combine them. The idea is that the axes are not affecting each other much (increasing the thrust is not likely to affect the rotation along any axis, for example), so it's safe to assume this will work. It reduces the original $O(pn^4)$ into much better $O(4pn)$.

On our hardware, the multiplicative coefficient change of $\frac{81}{12}$ at the smallest range (where $n = 3$) turned the AI from major bottleneck into something that took significantly less computing time than graphics module (which dictates the upper frame rate limit anyway). It is also recommended way of solving the inverse motion problem in the Flying Samurai game, producing easily achievable and solid results for most situations.

Iterative search Another way to improve raw speed (but when the precision and reproducibility suffers) is to use random hill climbing approach. This algorithm will take a random coordinate out of the four (thrust, pitch, roll and yaw) and changes it to another random value in the $[-1, 1]$ (resp. $[0, 1]$). Then it compares the performance to the previous state. If it has improved, it forgets the old configuration and replaces it with the newly altered one. This iterative process is repeated for the specified number of steps. The computational complexity goes down to $O(np)$, where n is freely configurable and signifies the upper limit of iterations. On the other hand, the performance is rather low, as it doesn't use neither gradient or any wider knowledge about the rather continuous nature of the world.

Neural network The most complex, but also the most promising method (if well trained, of course). It utilizes traditional neural networks with the input layer representing the decision values structure and ability to read the input values on the output layer[1]. The time to retrieve an answer from the network is basically constant (depends only on the amount of neurons, which in all practical uses is comparatively low, otherwise will hardly converge during the training phase anyway).

To train the neural network, one can use the `invphysics.exe` tool, which reads the `Entities.xml` file the same way the game would read it (see the appendix for the file's syntax – it describes also the tool's learning parameters) and perform a blend of back-propagation algorithm and evolutionary selection.

This approach have shown after few days of guided training (that is, altering the learning parameters by hand when appropriate to speed up the whole process) a reasonable improvement over the baseline (freshly initialized network). In many cases, it even could over-perform the brute force (as the brute force is still limited by the size of the step), but these results were scarce and also the network wasn't performing well in all possible situations the plane can get into.

Combination Obviously, there are positives and negatives about all the previous methods. In strive for being able to use not completely trained neural network, but still being at least partially stable, we also implemented a combiner of the previous inverse motion algorithms. As every algorithm is able to score itself (using the difference function in `PhysicsPattern` class derived from an abstract `NeuralPattern` – contrary to the name usable also in other methods), we can let two or more of the previously described ones let make its decision and then just pick the best one. In practice this is accomplished by multiple inheritance from the other classes, where the core method calls one-by-one the inherited core methods, keeping track of the best achieved score. That one is returned.

[1]The author is aware this usage of "input" and "output" is a great source of confusion. But inverse physics' main purpose is to convert a value that is an output of the traditional physics into a value that given as an input will provide the mentioned output. For this simple reason, there is little choice of different terminology.

4.8 Input

Our game is supposed to be a realistic flight simulator. This implicates several requirements on input. First, digital (two state) input is not enough and analog input has to be supported. Second, the airplane (usually) has 3 degrees of freedom, which means there have to be (at least) three different airplane controls governed by the input system. Finally, since joystick is the preferred input method for most flight simulation players, joystick support was almost a must.

Our library of choice is OIS – the Object-oriented Input Library – which interfaces the input hardware, including joysticks, nicely.

We chose to provide a query-based, rather than callback-based, interface, since queries can be made more than once, which allows different threads to make the same query in a frame's timespan. If callbacks were used, they would have to broadcast the information, and the callbacks would have to be triggered at before the frame starts but after the swap has been made, so that the handlers would already have access to the fresh readers and writers (see Section 4.2). Currently, input is polled (see `Poll`) during the swap and is guaranteed to return the same values over the frame's timespan. This might, however, cause problems when the game is running at rather slow framerates, since some keystrokes, for example, might get missed.

4.8.1 Abstraction layer

Since OIS works in a rather low-level fashion, a further abstration layer was needed. Ideally, the physical module shouldn't be concerned with whether the input came from the joystick, the mouse, or the keyboard. To provide some level of input configuration, a mapping from the hardware input (joystick position has been changed, key has been pressed, etc.) to in-game actions (roll to the left, yaw to the left, pitch up) has to be provided.

The hardware input comes in three different flavours. A key or a mouse button is either pressed or released, but the joystick position is given in absolute coordinates (this is device dependent, but it was so for the joystick we tested with), and mouse movement is best tracked by checking it's relative coordinates (with respect to last frame; if we used absolute coordinates instead, mouse movement wouldn't change the mouse position when the cursor is located at the edge of the window, whereas relative coordinates work fine in this case).

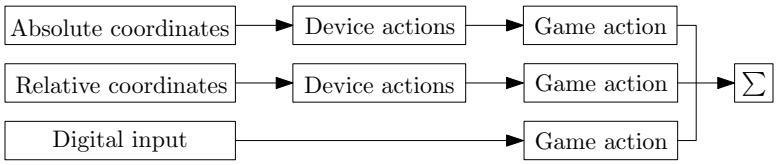


Figure 23: The input pipeline, showing the data-flow for each kind of input.

Next, having a position doesn't really tell us much about what has actually happened. The action (movement to the left, for example), has to be extracted from the raw coordinates. We call those actions *device actions* (see `MouseAction`, `JoystickAction`,). Note that the keyboard, being a two-state device, doesn't have this problem and no actions need to be explicitly defined (they are already predefined by OIS and are identical to the key-codes).

Finally, if the user uses all possible input devices at the same time, possibly even triggering the same action using different input devices, which of them should we prioritize? We chose to sum all of the influences (we call them *amounts*) together (see `GetAmount`). Therefore, for each device, a mapping between de-

vice actions and the game actions (see `InputAction`) is defined (see `m_keyBinding`, `m_mouseBinding`, and `m_joystickBinding`).

Since the keyboard is a two-state device, a default amount for each of it's actions has to be defined. We hoped to implement an input aid that would make using the keyboard easier by changing the amount continuously depending on how long the action has been pressed or released (see below; a similar system is sometimes implemented in car simulations). This, though, proved not to be necessary.

For some actions, it is also useful to track whether the action has just started, is in progress, or has just ended. We chose to re-implement this functionality in the abstraction layer, even though OIS already provides it. By keeping track of the previous frame (see `m_inputFrame`, `m_currentFrame`, and `m_lastFrame`), we are able to answer the above-mentioned queries (see `HasBeenPressed`, `HasBeenReleased`, `IsPressed`).

Special handing of the `SHIFT` key was needed to compensate for a known bug in OIS-CEGUI integration. Extra mouse keys also need to be filtered to avoid crashes.

4.9 Sound

4.9.1 Requirements

The Game takes place in a three dimensional environment and sound effects should make use of this to provide a more authentic experience. The engine used therefore has to be able to "position" each track/sound in the 3D environment. Support for the Doppler effect and customized sound roll-off decided that iirKlang [15] was chosen as the most suitable engine. The engine itself supports multi-threading, 3D sound, the Doppler effect, customized playback, and is very easy to use.

4.9.2 Resources and usage

The game provides these sound effects:

- **Engine sound** - each plane emits thrust dependent noise.
- **Fire** - emitted by each machine when a bullet leaves its barrel.
- **Explosion** - when the plane collides with the ground or with another plane.

The pitch of the engine sound is modulated according to the level of thrust each plane is currently at (see `GetSoundCoefficient` and `GetSoundVolume`). Since engine is quite loud, it is to be expected that most of the time, the player will only be able to hear his own engine and only close proximity to another plane makes location by sound possible. Listening point position is determined by the user's position vector, direction vector, orientation vector and velocity vector(for Doppler effect). Engine and bullet sounds require position vector and velocity vector (for Doppler effect). Explosions are played as 2D sounds. These help to identity whether a airplane crash occurred.

4.9.3 Integration

The author of the irrKlang library recommends the sound interface to run either in a separate thread or to receive updates roughly thirty times per second. The multi-threaded mode allows the usage of irrKlang in its own separate thread. To avoid concurrency problems, we used the default multi-threaded mode with a re-entrant interface.

The engine itself is represented by a `ISoundEngine` instance. During start-up, game pre-loads all sounds to memory for uninterrupted playback. Every airplane has it's own paused instance of engine sound. Every scene graph update triggers the refresh of engine and bullet sound positions. Sounds are paused when the user enters the in-game menu. Sounds are resumed when the user returns to the game. Game does not play any sound during start-up and when in menu. Pausing and resuming is triggered in logical action handler in `GameLogicModule`

4.9.4 Interface

The game wraps the sound in the `Sound` class. This class provides functions to add 3D/2D sound using Ogre3D vectors. Bullets and explosions use non-looped tracks and the interface does not track their playback. Engine sounds are played in a loop and the wrapper keeps their handles to allow for pause and resume functionality. These handles are mapped to `s32` sound indexes in order to allow for easy sound engine exchange.

4.10 Miscellaneous

Information about those pieces of code that do not directly fit into any of the other sections or that are span multiple section with no clear affinity can be found here.

4.10.1 Consoles

To make debugging easier, several debugging consoles were introduced. These convey information about the state of various parts of the game. Because of the reader/writer paradigm, the consoles have to be handled in two separate classes, the `Console` and `GraphicalConsole`. The `Console` class implements both the reader and writer via the !`ConsoleReader` and `ConsoleWriter` classes and therefore buffers the data to be displayed. The `GraphicalConsole` class then takes care of rendering the data from the `ConsoleReader`.

Both classes work with lines. Each line has a number and the data can be changed on a per-line basis. While lines with newline characters display correctly, the lines below them have no concept of that and will overwrite a part of the line with newline characters. As a rule of thumb, either one single line with newline characters or several lines without newline characters should be used.

The `GraphicalConsole` class is based Ogre3D's `Overlay` and `PanelOverlayElement` classes and uses the default material from Ogre3D demos for console background. Screen position in $\langle 0, 1\rangle \times \langle 0, 1\rangle$ can be specified for each of the corners.

It is worth noting that the text output in Ogre3D is rather slow and displaying too much textual data might slow the game down considerably.

4.10.2 SettingsFile

With the exception of `Entities.xml`, all data files are read and written by the `SettingsFile` utility class. Class is essentially a strong typed key-value pair storage. Each module is provided with its own settings file. Career and statistics data are also serialized and de-serialized using the `SettingsFile` class.

Usage:

```
SettingsFile settingsFile;

if(!settingsFile.Load("superman1.dat"))
                throw std::exception("too bad");

if(settingsFile.HasKey("vertical_resolution"))
                u32 verticalResolution =
                    settingsFile.GetU32("vertical_resolution");

if(settingsFile.HasKey("horizontal_resolution"))
                u32 horizontalResolution =
                    settingsFile.GetU32("horizontal_resolution");

// add some other key
settingsFile.AddBool("fullscreen", true);

settingsFile.Save("superman2.dat");
```

Later in development, we added support for container storage. Using the `Checksum` class, we assure the

integrity of career data. The user can only load careers that have been created with the current `Entities.xml` file.

4.10.3 Command line parameters

The game supports various command line options. We use `boost::program_options` since it seems to be the most elegant solution available in the currently used 3rd party libraries. This mechanism is used to easily add simple development related functionality

- `--settings-path <dir>` - path where the Startup (see Section A) settings file is located. If not provided, the current directory is used as a default.
- `--simple-tests` - runs tests that only take a short amount of time upon game start.
- `--run-all-tests` - runs all tests upon game start.
- `--disable-terrain` - replaces terrain with a simple plain terrain with uniform height. Used for mainly in debug mode to make the game load and run faster.
- `--help` - prints command line options to console.
- `--console` - shows external command line console window with debug output.
- `--log-file <file>` - redirects standard output to the specified file. Console window output is preserved.

  Please note that unit tests were only used to debug utility classes in early stages of development. It is not guaranteed that tests will pass or even be able to finish without a crash, since they are not up to the date with the rest of the code.

4.10.4 Helper code

Game code interconnects several frameworks and their structures are naturally not compatible. A good example of this is the conversion of Bullet, Ogre, and irrKlang vector types. We implemented conversion functions to seamlessly pass data from one framework to another. There was also the need for basic type conversions (from string and to string). `ToString` and `FromString` template functions are used for this purpose.

Although it is considered a bad practice, we use C macros to simplify or remove redundant code. This is used in shared structures(`FS_BUFFERABLE` and `FS_BUFFERABLE_EX`) and entity classes (`EntityWithMap`, `EntityWithList` and `EntityWithList2`, see Section 4.2.5). Macros are an essential part of the transparent memory tracker.

We have also `typedef`ed most of the common built-in types to make the usage of signed and unsigned types, as well the use of specific length types, easier; see `Types.h`.

4.10.5 Code sharing foundation

A Subset of header and `cpp` files is prepared for usage in multiple projects. This can be seen in invphysics and game projects. It is accomplished by using compatible pre-compiled headers and project specific C `#define`s to disable unused code with `#ifdef`s. The most heavily affected file is `Globals.cpp` and it illustrates how only some modules are used in each project.

4.10.6 Multi-platform support

The game can be currently run on the Windows platform only. There are no known problems that would not allow the project to be built (but not run!) on a platform intersection of used 3rdp arty libraries. If someone would attempt to port code to different platform and/or different compiler, the following needs to be done:

- Ogre3D library and its plugins must build manually from supplied patched sources.
- `game_pch.h` contains a few parts that have to be modified.
- We use custom CMake include files and these have to be modified.
- The OpenGL renderer in Ogre3D stopped working early in the project's development. All of our shaders have been written in Cg[3] (we have also used some of Ogre3D's default materials that have been written in different shading languages) and we believe this or the usage of some OpenGL incompatible construct in Cg is the source of the problem. We have not investigated into the matter any further, though.

5 Conclusions

After over 20 endless months of work, we have approximately gotten to the point where we wanted to be when we started. Of course, the game is not an AAA title, and there are still details that we could fix and features that could be added, but the result is very satisfying and, especially in comparison with professional projects such as Rise of Flight and Over Flanders Fields (see Section 2.5), we are very proud of our achievement. The first feedback we have gotten from the community (at the time of writing, the game hasn't been released yet) is also positive.

It goes without saying that we have all improved our (or gained our first) teamwork skills. We have all seen and experienced how important tolerance, work morale, covering each others back, team spirit, project management, and individual skills are. Several unexpected events – especially the unexpected crash of the supposedly backed up `urtax` server that hosted our SVN/SVNStats and DokuWiki – have also shown us that risk management shouldn't be underestimated. Working with externists has also taught us a lot about the motivation in other people and increased our management skills.

The sheer size of the project forced us to integrate and adapt several already existing solutions, something one doesn't usually encounter much while working on his/hers lab assignments. Our ability to read and understand other people's code, as well as the ability to write code that can easily be understood, has also improved.

While several non-ideal decisions were made over the course of the project, most of them were forced or couldn't have been avoided without further experience. If we were to do the project over again, we'd probably drop the paged terrain solution which was very hard to get running and maintain in favour of a more simple – possibly non-paged – one, and drop and replace the CEGUI library, which has proved to be rather difficult to use well.

All in all, we believe we can hold our heads up high.

6 Acknowledgements

First, we would like to thank our project supervisor, Mr. Otakar Nieder, for his responsible attendance at our weekly project meetings, for his help with preparations of the airplane meshes, and of course for his share of leadership.

Secondly, we thank all externists that participated on the project by doing historical and technical research, game content creation, and testing. Their names and respective contributions to the project are mentioned in Section 2.2.

Thirdly, we thank the developer teams of all the libraries, plugins, and software we have used – Ogre3D graphics engine, Bullet physical engine, Opensteer library, Boost library, irrKlang sound engine, tinyxml library, Myrddin Landscape plugin, PagedGeometry plugin, CEGUI library, HDRlib plugin, L3DT software, and probably others.

Last, but not least, we would like to thank Lenka Forstová for providing us with the room for the project meetings and a PC workstation for overnight neural network training, Tomáš Holan for consulting the administrative side of the project with us, and APS Group, s.r.o. for hosting our TRAC and SVN repository (after the crash of the `urtax` server).

7 References

[1] Bloom, Charles: Terrain Texture Compositing by Blending in the Frame-Buffer, http://www.cbloom.com/3d/techdocs/splatting.txt, 2000

[2] CEGUI plugin for Ogre3D, by Crazy Eddie and open source community, http://www.cegui.org.uk

[3] Cg - the Language for High-Performance Real-Time Graphics, http://developer.nvidia.com/page/cg_main.html

[4] CMake, by Kitware, http://www.cmake.org

[5] DirectX library, by Microsoft Corp., http://www.microsoft.com/windows/directx

[6] DotScene fileformat for Ogre3D, http://www.ogre3d.org/wiki/index.php/DotScene

[7] Flying Samurai: Specifikace softwarového projektu, http://ksvi.mff.cuni.cz/~holan/SWP/zadani/flying.pdf

[8] Glasser, Nate: Texture Splatting in Direct3D, in gamedev.net, http://www.gamedev.net/columns/hardcore/splatting

[9] Google Earth software, by Google, http://earth.google.com

[10] Globally Unique Identifier (GUID), in Wikipedia the Free Encyclopedia, http://en.wikipedia.org/wiki/Globally_Unique_Identifier

[11] HDRlib plugin for Ogre3D, by Christian Luksch, http://www.ogre3d.org/wiki/index.php/HDRlib

[12] Heightmap, in Wikipedia the Free Encyclopedia, http://en.wikipedia.org/wiki/Heightmap

[13] How To: Generate Superb Heightmaps, in Transport Tycoon Forums, http://www.tt-forums.net/viewtopic.php?f=29&t=27052

[14] Inkscape, by open source community, http://www.inkscape.org

[15] irrKlang, by Ambiera, http://www.ambiera.com/irrklang

[16] L3DT software, by Bundysoft, http://www.bundysoft.com/L3DT

[17] MICRODEM software, by Peter Guth, http://www.usna.edu/Users/oceano/pguth/website/microdem/microdem.htm

[18] Myrddin Landscape Plugin for Ogre3D, by Jacques Quidu, http://myrddinplugin.sourceforge.net

[19] Ogre 3D engine, by Torus Knot Software and open source community, http://www.ogre3d.org/

[20] OpenSteer library, initially developed by Craig Reynolds, Research and Development http://opensteer.sourceforge.net/

[21] Over Flanders Fields mod for Microsoft Combat Flight Simulator 3, by OBD Software, http://www.overflandersfields.com

[22] PagedGeometry Engine for Ogre3D, by John Judnich, http://www.ogre3d.org/wiki/index.php/PagedGeometry_Engine

[23] Photoshop, by Adobe, http://www.adobe.com/products/photoshop/compare

[24] Red Baron II, in Wikipedia the Free Encyclopedia, http://en.wikipedia.org/wiki/Red_Baron_II

[25] Red Baron II/3D, by Dynamix, http://www.wingsofhonour.com/redbaron3d/html_woh_redbaron3d_about.en.html

[26] Rise of Flight, by neoqb, http://riseofflight.com

[27] Strauss, Paul S.: A Realistic Lighting Model for Computer Animators, in proceedings of IEEE Computer Graphics and Applications, 1990

[28] Truevision TGA, in Wikipedia the Free Encyclopedia, http://en.wikipedia.org/wiki/Truevision_TGA

[29] VC++ 2008 SP1 Compiler, by Microsoft Corp., http://www.microsoft.com/downloads/details.aspx?familyid=a5c84275-3b97-4ab7-a40d-3802b2af5fc2&displaylang=en

[30] Windows SDK 6.0, by Microsoft Corp.

[31] Wings of Honour, Rise Of Flight - Interview with neoqb, http://www.wingsofhonour.com/riseofflight/articles/interview_neoqb_20090813/html_woh_riseofflight_interview_neoqb_20090813.en.html

A Startup

To provide customized application launch, a .NET launcher was created. It main function is to provide the game with a graphics and sound configuration dialogue (we wanted to replace Ogre3D's configuration window, as it contained some options that could crash the game, such as the OpenGL renderer). The Startup application (see Figure 24) replaces this mechanism and among other things, shows our own logo.

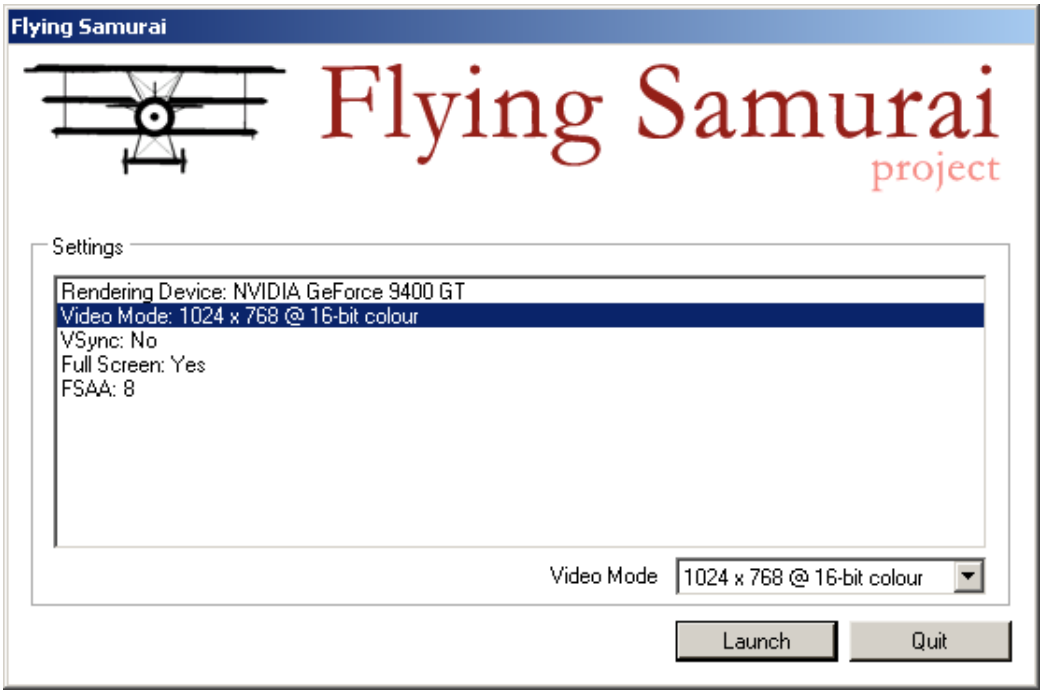


Figure 24: The Startup configuration dialogue.

User configurable items are fully customizable. The assembly references DirectX binaries and inter-op binaries to detect available video modes. The configuration file is then saved to the user profile directory. The user interface roughly copies the behaviour of the Ogre3D configuration dialogue. The configuration file is loaded and stored using serialization. After the user decides what graphics and sound settings he wants, the "Launch" button starts `game.exe` and makes it read the user specific configuration file.

B Game parameters definition file ("entities file")

This section describes the "entities file". The purpose of this file is to define parameters and objects for the game logic, such as airfield positions, airplanes, pilots, nationalities, etc. Because the file is an XML document, basic to intermediate knowledge of XML is assumed.

Basic concepts

Two basic concepts need to be understood. First of all, some entities in the file might be cross referenced. A cross-reference defines a relationship between these two entities, for example a pilot might cross-reference an airplane (*"I'm the pilot of that airplane"* relationship). For this purpose, ID numbers are used and the entities then mention the ID of the other entity somewhere in their definition.

Second, some of the entities can change their attributes with time. This is achieved using the `From` attribute. This attribute specifies the date from which the set of values is valid. There is no `To` attribute, once the date in the campaign goes far enough, a definition with a more recent `From` attribute will be used. The concept is illustrated on the following example:

```
<Front From="1910-01-01"> .. </Front>
<Front From="1912-01-01"> .. </Front>
```

Two sets of data for the front are specified, one valid from 1. January 1910, one valid from 1. January 1912.

B.1 MOD

The entity file first defined basic properties of the current game MOD. This is done inside the ¡Mod¿ tag as follows:

```
<Mod Name="Original Mod" From="1914-08-28" To="1914-09-10"/>
```

Name attribute should be a unique string identifier, `From` and `To` attributes specify the beginning and end of the campaign. The dates above are valid for the World War I., but can be arbitrary dates in the `YYYY-MM-DD` format.

B.2 Period

The campaign might be divided into different periods, each of them specified by a `<Period>` tag. The goal of this option is to provide a way to control the frequency of missions (less missions at the beginning of the war) and the type of the missions (less bombing missions at the beginning of the war). Note: The mission types are fixed (cannot be redefined)!

```
<Period From="1914-08-28" Missions="3" SquadronDogFight="1" PatrolTheFront="2"
    EscortBombarders="0" RaidBombarders="0" />
```

The `From` attribute specifies the time dependency. Missions attribute specifies the number of mission in that time period (until the next time period takes over). The `SquadronDogFight`, `PatrolTheFront`, `EscortBombarders`, `RaidBombarders` attributes specify the ratio of the various mission types during that

time period. For the above mentioned example, the ratio would be 1:2:0:0, i.e. some squadron dogfights, twice as many patrol the front missions and no escort bombers or raid bombers missions.

B.3 Map

The Map tag defines properties for the map that is to be used. Currently, the map of western Europe is always used, it's width and height in meters is defined using the `Width` and `Height` attributes of the map tag

```
<Map Width="530000" Height="430000" />
```

B.4 Front

The front is time dependent. The `<Front>` data are valid from the date specified by the `From` attribute and until either the end of the war, or until overridden by a next `<Front>` tag with a newer date.

```
<Front From="1910-01-01">
...
</Front>
<Front From="1912-01-01">
...
</Front>
```

The above example defines two fronts, one valid from 1. January 1910, and another one, valid from 1. January 1912. This way, the shape of the front can change with time. Note that the `From` attribute can contain dates that are less recent than the start of the MOD.

Inside the `<Front>` tag, the actual front data are given. A set of points is given, consecutive points being connected by a line. Each point is given as a `<Point X="0" Y="0" />` tag, where the `X` and `Y` attributes define the position of the point on the map (in meters; relative to the upper left corner, `X` going from 0 (left border of the map) to map width (right border of the map), `Y` going from 0 (top of the map) to map height (bottom of the map).

```
<Front From="1910-01-01">
   <Point X='265000' Y='0' />
   <Point X='269755' Y='8600' />
   <Point X='267939' Y='17200' />
   <Point X='262062' Y='25800' />
</Front>
```

B.5 Nations, Sides

The game recognizes sides and nations. There are always two sides only (for WW1, that is the allies and the entente powers). Each side might have one or more nations, i.e. Great Britain and France for the allies, Germany and Austria-Hungary for the entente powers. Sides and nations could be specified in the following way:

```
<Side ID="1" Name="Allies" />
<Side ID="2" Name="Entente" />
<Nation ID="1" Name="Great Britain" SideID="1" />
<Nation ID="2" Name="France" SideID="1" />
<Nation ID="3" Name="Austria-Hungary" SideID="2" />
<Nation ID="4" Name="Germany" SideID="2" />
```

Notice that the nations cross-reference the sides they belong to by using the `SideID` attribute. Name and ID obviously specify the unique identifier and the name of the respective side/nation.

B.6 Airfields, Squadrons

Airfields and squadrons are quite similar. Each `<Airfield>` tag defines a physical airfield. Each airfield might be shared by one or more squadrons. The usual pattern would be to first define an airfield, and then define one or more squadrons that would occupy this airfield. Airfields are specified in the following way:

```
<Airfield ID='33' Name='33_axis' NationID='2'>
   <Location X='267568' Y='409898' From='1910-01-01'/>
   <Location X='267768' Y='409298' From='1912-01-01'/>
</Airfield>
```

The `ID` is a unique identifier of the airfield and will be used elsewhere to reference this airfield. The `Name` attribute specifies the name of the airfield, (the name should probably change as the location of the airfield changes; this is currently not so, choose a generic name like the one above if more names would apply) `NationID` references the nation this airport belongs to (not side!). Inside of the `<Airfield>` tag, locations that change with time can be defined, `X` and `Y` attributes specifying position of the airfield from a given date, `From` specifying that date. Squadrons are defined like this:

```
<Squadron ID='0' Name='Flying Circus'>
   <Airplane AirplaneID='1' From='1910-01-01' />
   <Station AirfieldID='0' From='1910-01-01' />
</Squadron>
```

`ID` attribute specifies the squadron's unique identifier, name defines the name of the squadron. The squadron's assignment to airfields is time dependent and can be specified using the `<Station>` tag, where the `AirfieldID` specifies id of the airport to assign this squadron to, and `From` specifies the date from which that should happen.

Each squadron might also have a default airplane. This assignment is also time dependent, as at the beginning of the war, different default airplanes will be used than at the end of the war.

B.7 Airplanes

Each airplane has to have it's own `Airplane` tag. The `ID` attribute specified the airplane's unique identifier. The `GUID` attribute specifies the (see Section 4.5.8). `FullName` attribute contains the airplane's name, `NationID` attribute specifies the ID of the country that build the airplane, `MountedGunID` attribute specifies the ID of the gun that is mounted on the airplane, `AIParams` specifies the set of AI parameter to be used, `BuildFrom` is that date the airplane's production started, `BuildTo` is the date the airplane's production ended, `AmountBuilt` is currently ignored, and finally, the `PhysicsID` attributes specifies the physics model to use. Within the `<Airplane>` tag, the `<description>` tag contains a character data section with the airplane's description that will be used in the career mode.

```
<Airplane
              ID="5"
              GUID="111"
              FullName="Fokker E III"
              NationID="2"
              MountedGunID="6661"
  AIParams="5"
              BuildFrom="1914-01-01"
              BuildTo="1920-01-01"
              AmountBuilt="100"
              Powerplant="100"
  PhysicsID="1"
>
  <Description>
    <![CDATA[Scourge of the air during the winter of 1915, the Fokker E.I was
        the first aircraft armed with a synchronized, forward firing machine
        gun. German pilots were ordered not to fly it across enemy lines for
        fear the Allies would capture the secrets of the synchronizing gear.
        Followed by the E.II, E.III and E.IV, the Eindecker was underpowered
        and slow but could out turn most of its opponents. Allied aviators
        who faced it called themselves "Fokker Fodder". The Eindecker ruled
        the skies until the Nieuports and SPADs were developed.]]>
  </Description>
</Airplane>
```

B.8 Pilots

Pilots are defined like this:

```
<Pilot ID='1001' FirstName='Manfred' Surname='Richthofen'
   DateOfBirth='1900-01-01'  DateOfDeath='1960-01-01' NationID='2' />
```

`ID` is a unique identifier and should be higher than $20 \cdot \#\text{nations} + 1$. It's advised to reserve about a thousand ID's for each nation, i.e. all Germans could be 1001, 1002, ..., all Brits could be 2001, 2002, ... etc. Lower numbers are reserved for dummy pilots (unnamed pilots that fly bombarders etc). `FirstName` is the first name of the pilot, `Surname` is the surname of the pilot, `DateOfBirth` specifies the date of birth for that pilot and serves no real purpose as of now (might be visible in the GUI at some later point in time), `DateOfDeath` specifies when that pilot dies. If that date is within the MOD's starting and ending date, the pilot will really die then (unless he gets shot down in a confrontation with the human player earlier), i.e. he will not take part in any missions after his date of death.

B.9 Ranks

Ranks are defined as follows:

```
<Rank ID="0" NationID="1" Name="Rittmeister" Downs="0" Missions="0" Average="0"
    LossAverage="1.0" Days="0" Score="0" CanPaint="false" CanTransfer="false"
    CanLead="true" />
```

`ID` is the unique identifier, `NationID` cross-references the nation this rank belongs to, `Name` is the name of the rank. `Downs` (number of enemy airplanes downed), `Missions` (number of mission flown), `Average` (average number of shot downs per mission), `LossAverage` (average number of lost friendly airplanes per mission), `Days` (days in service), and `Score` (achieved score) define the requirements necessary for the player to achieve that rank. `CanPaint` (can paint his airplane, i.e. Red Baron, currently ignored), `CanTransfer` (can the holder of this rank request transfer to some other squadron?), `CanLead` (can the holder of this rank select formation, number of airplanes etc. before each missions?) define the qualities of this rank. `LossAverage` might be useful to discard players who lose too many friends in battles, a good leader would never let that happen. Other conditions are pretty self-explanatory in their use.

B.10 Awards

For each nation, a set of awards can be defined. An awards is defined in the following way:

```
<Award ID="0" NationID="1" Name="Pour le Merit" Downs="17" RankID="3"
    Missions="0" Average="50.0" N3="1" N4="2" N5="3" Score="100"
    Posthumously="true" Description="Pour le Merit is one of the most
    prestigious..." />
```

`ID` is the unique identifier, `NationID` cross-references the nation that awards this awards, `Name` is the name of th awards. Following attributes define the condition under which the award is awarded: `Downs` is the minimum number of downed enemy airplanes, `RankID` is the minimum attained rank, `Missions` is the minimum number of mission flown, `Average` is the shot-down average per mission, `N3` is the number of times 3 or more airplanes were downed by the pilot during a single mission, `N4` is the number of times 4 or more airplanes were downed by the pilot during a single mission, `N5` is the number of times 5 or more airplanes were downed by the pilot during a single mission, `Score` is the minimum total score achieved by the player. Finally, `Posthumously` defines whether the award might be awarded posthumously and `Description` is a longer description of the award.

As a guideline, awards where big bravery is required might use `N3`, `N4` and `N5` to require the player to down several enemy airplanes during one mission. `Rank` might be used to limit the recipients only to some level of ranks, `Missions` might be useful for "lifetime achievement" awards or to require a certain experience before the award is awarded, `Average` might help you to distinguish between experienced players who do okay, but who generally don't perform above-par; in combination with the `Missions` attribute, this will be useful for awards that shouldn't be achievable just by pure endurance, but that require a long-term above-par performance level.

B.11 News

Various news can be specified and will be shown to the user at the first occasion after the given date (when in campaign mode). These can be defined as follows:

```
<News Date="1917-03-10" Message="The allies now dominant" ImagePath="path" />
```

`Date` attributes specifies the date in a `YYYY-MM-DD` format, `Message` is the message to show, `ImagePath` attribute should contain path to the image that is to be shown (the `ImagePath` attribute is currently ignored)

B.12 Physics, Gun

Every physics section bears the physics parameter set for an airplane. It should be specified before the Airplane tag that uses it. More details on those sections can be found in 4.6.3. In addition to the described attributes, gun contains the `FullName` and `GUID`. Finally, both physics and gun tags have an `ID` attribute that is used solely for purposes of binding it with the corresponding airplane.

B.13 AIParams, NeuralParams

These two tags specify the parametrization of the AI module, either directly the parameter set affecting the agent behaviour or the neural network used in the inverse motion. The attributes of `AIParams` (all are required) consist of

Inverse specifies which algorithm should be used for inverse physics. Valid values are `Brute`, `ByAxis`, `Iterate`, `Neural` and `Combine`.

UseNeural a boolean value used in case inverse physics algorithm is combining.

UseBrute a boolean saying whether to include brute force algorithm in combining inverse physics algorithm.

UseIterate a boolean saying whether to include iterative algorithm in combining inverse physics algorithm.

Smoothing is a boolean saying whether the actual output of the inverse motion algorithm should be a weighted average of the previous control position and the new one.

SmoothingCoef is a coefficient for weighted average of the previous input value set.

Thrust/Pitch/Roll/YawStep are settings for the brute force search in the input space.

Thrust/Pitch/Roll/YawRange are ranges for the brute force search in the input space.

MaxIterations is a parameter for iterative inverse motion algorithm.

NeuralParams is an ID number of the (previously specified) neural parameter set.

Individual is the number of the individual in the population that will be used for neural network inverse physics algorithm (the first indexes belong to the elite).

WideAngle specifies the cosine of an angle in which the turning towards the prey will take place.

NarrowAngle specifies the cosine of an angle inside which shooting makes sense.

FriendlyFireAngle defines the cosine of an angle in which there should be no friendly airplane (otherwise the shooting is prevented).

PreyClose, PreyFar define parameters for the *catching* agent state.

MaxSpeed, MaxForce define limits for OpenSteer calculations. Lower values will force AI to somehow limit its manoeuvre skills.

And the neural params are as follows:

InputGenerate new input sets (in the size of `FitnessTests`) are generate every n generations.

LongTerm says how many generations are considered long-term (for on-screen statistics output).

MaxGenerations specifies when to stop the evolution.

SaveInterval the progress (binary genome of every individual) is saved every n generations.

DoReverts whether to revert to a previously saved population if the performance decreases.

ReportRevert whether to write an on-screen report about the revert.

ReportMilestone whether to report a milestone (for revert purposes).

Population number of individuals in population.

Layers number of hidden layers (that is, excluding the input and output layer).

LayerSize the size of every hidden layer.

FitnessTests how many tests are performed every generation.

Mutation probability to mutate.

Recombination probability to recombine.

Education probability to perform a back-propagation learning.

LearnRepeat states the number of repeats the back-propagation will be performed.

Elite is the number of elites in population.

Garbage is the number of worst individuals in population that will be thrown away.

Injection is the number of new individuals introduced every generation.

CodeTolerance what weight at the synapse will be tolerated without penalty.

CodeSeriousness the multiplicative coefficient for the penalty derived from overly high synapse weights.

MutationRange the range for floating point mutation.

InitialRange the initial range of weights for new individuals.

Lambda learning coefficient.

C Media directory

This section describes the structure of the `Media/` directory. We use this directory for storing all multimedia game content (mainly related to the Ogre3D engine), as the name suggests. Also some configuration files are placed in this directory.

`Media/`	The root directory of the tree. Also contains `OgreCore.zip` archive (contains few Ogre3D minimal-profile objects), `forestdata.zip` archive (forest density maps), `Objects.xml` config file (a list of registered game objects, mostly airplanes) and `land.xml` (MLP terrain configuration file).
`Media/citydata/`	Building positioning data tiles (see Section 4.5.4).
`Media/debugging/`	Textures and other resources related to various debugging and helper features.
`Media/flora/`	Media related to tree models (currently only a single tree).
`Media/gui/`	Resources related to GUI display, such as layouts, schemes and underlying textures.
`Media/hdrcompositor/`	Subtree used by the HDRlib plugin for fullscreen bloom and HDR effects.
`Media/mlp_terrain/`	Resources related to terrain rendering and Myrdding Landscape Plugin.
`Media/mlp_terrain/hf/`	Skeleton terrain representation. Contains two archives, `hf.zip` and `hf_sm.zip`, which contain terrain elevation data and terrain splatting maps, respectively. See Section 4.5.3.
`Media/mlp_terrain/materials/`	Resources that influence terrain appearance, such as material scripts and programs, splatting textures and other textures.
`Media/models/`	Mesh entities. Mainly contains airplanes and their textures (resources of each airplane are stored in a special subdirectory of this directory, named as the respective airplane).
`Media/programs/`	GPU programs, most of them written in Cg and HLSL languages.
`Media/scripts/`	Material, compositor and particle script files, written in Ogre3D's dedicated language.
`Media/sound/`	Sound files in `.wav` format, used by our experimental sound system.
`Media/textures/`	Several image textures that don't belong to any other category of entities.
`Media/textures/sky_polar`	Hemispherical sky maps in polar coordinates. To add a new one, the given naming must be obeyed and the corresponding setting in `graphics.cfg` must be changed to reflect the new number of skymaps.

D Terrain preparation workflow

This section describes the process of preparation of the data which are related to the game terrain. Note that the structure and usage of these data is described in Section 4.5.3 and Section 4.5.4; this section only describes how these data are obtained and what processing of them has to be performed to produce a form readable by the game engine.

We use several third party applications to manipulate and process these data. These are namely *Adobe Photoshop* [23] (raster image editor), *Google Earth* [9] (3D Earth viewer), *MicroDEM* [17] (GIS application for processing digital elevation maps), *L3DT* [16] (synthetic terrain generator and elevation data editor) and *Inkscape* [14] (vector graphics editor). All of them are freely available, except Adobe Photoshop, which can be substituted by another raster image editor capable of batch-processing of files. We also use our own auxilliary applications, namely *TileCombiner* (a tool for manipulation with .tga tiles and operations over them, see Appendix H) and *CityGen* (a tool for generation of cities and road networks, see Appendix I).

As described in Sections 4.5.3 and 4.5.4, the terrain is composed of 4 types of data – heightmaps (elevation data), splatting maps (colouring and texturing data), and forest and city description data (localization of forests and buildings over the terrain). Since our terrain spans an area of roughly 200,000 km^2, it would be impossible to create all this content by hand. We therefore decided to generate most of these data automatically. This approach of course produces a terrain with a certain degree of uniformity, but at least makes incorporation of such huge landscape feasible. The advantage of our generation pipeline is that all these data are generated from real elevation maps obtained from the area where air combat during the World War I. was present (i.e. the Western front; but our engine is not limited to this area, any other set of data can be used in our game). And except elevation maps, no other data are necessary to generate the game terrain (just metadata about character of the country etc., which are used to configure various parts of the pipeline).

The following sections describe the preparation pipeline of all types of data: elevation maps (Section D.1), splatting maps (Section D.2), and static terrain geometry (Section D.3).

D.1 Terrain elevation maps

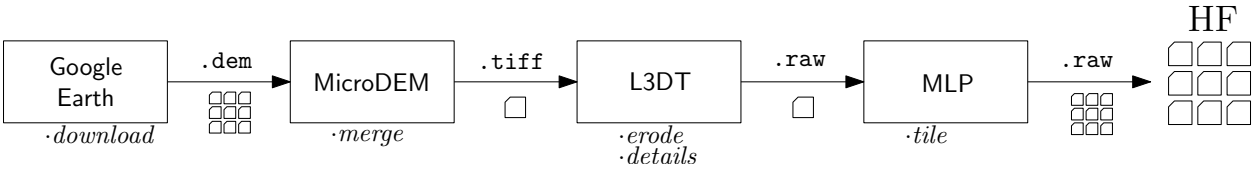


Figure 25: Heightmap creation pipeline.

This part of the pipeline if schematically shown in Figure 25.

1. Raw terrain elevation data in the .dem format are obtained from Google Earth (it is of course possible to use any other source of elevation data). The procedure for this, along with the next step, can be found at Transport Tycoon Forums [13]. In our case, we downloaded 4 .dem tiles of the area around the northern French border.
2. The .dem tiles are fed to the MicroDEM application, where they are processed and merged into a single elevation map in the .tiff format.
3. The heightmap is used as a basis for the L3DT application, where further processing of it might be performed. On our case, the resulting elevation map after processing in MicroDEM had the resolution

of about 8k×8k, which was insufficient for us. Therefore we used L3DT to rescale the heightmap to 16k×16k, on which we applied a simulation of erosion processes on the terrain. We then converted the elevation dataset to the 16bpp .raw format. Generally, these steps are not necessary, if the obtained dataset has sufficient details, but usage of L3DT (or some other alternative software) is recommended, as the heightmap is also used to generate some of the layers for the splatting maps (see Section D.2).

4. Finally, the Myrddin Landscape Plugin (MLP) has to be used to split the large singular .raw file onto smaller .raw tiles, which will be utilized by the paging mechanism of MLP during the rendering. This is done by calling `TerrainSceneManager::setOption("ToolGenerateMosaic", ...)` method (directly in code), specifying the appropriate parameters according to the documentation of MLP. For this, MLP has to be compiled with `TSM_EDIT` flag enabled. In our case, we divided the large 16k×16k heightmap onto an array of 32×32 tiles, each having the resolution of 513×513 (with 1-pixel overlaps). The naming of these files is according to the conventions of MLP, with `hf_` prefix.

D.2 Splatting maps

This part of the pipeline if schematically shown in Figure 26. This is the most complicated part of the terrain workflow. It consists of two branches – **L0** and **L1**, which are in the end combined to create the resulting splatting maps.

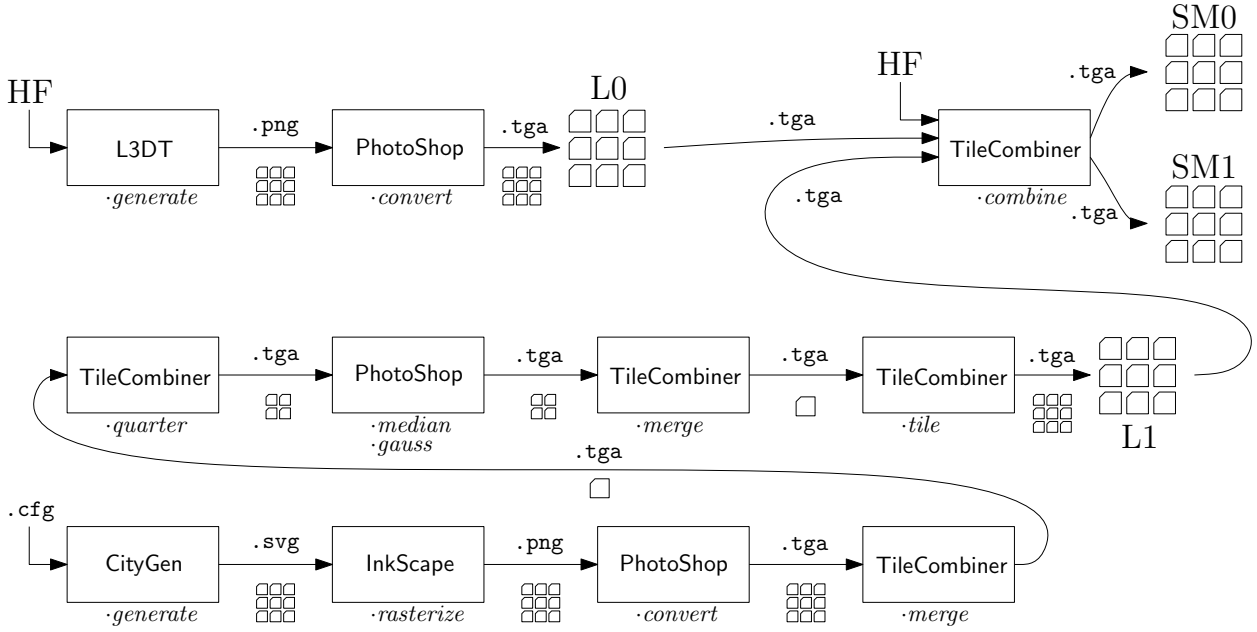


Figure 26: Splatting maps creation pipeline.

L0 branch This branch creates the 'natural' splatting layers – 2 grass types, rocks and snow.

1. L0 layers are created from the elevation maps during their processing in L3DT (as a side product on demand; they are designated as 'alpha maps' here). The underlying algorithm decides which material layer is present on a particular position based on the terrain elevation and slope, and probably other indicators. Since L3DT usually produces more than 4 required layers, it's necessary to fuse some of them, preferably the ones with low terrain coverage. The resulting data files are then exported as .png tiles (32×32 in our case, each having 1025×1025 texels).

2. These .png tiles are then batch-converted in Photoshop to uncompressed 32bpp .tga format, to be processed further in the TileCombiner application.

L1 branch This branch creates the 'urban' splatting layers – roads, fields and forests (hardly being 'urban', but anyway).

1. The initial representation of roads, forests and fields is generated by the CityGen application. Building data are also generated in this process (see Section D.3). The result depends on the CityGen configuration, consisting of a set of abstract parameters describing the target dataset. CityGen outputs an array of .svg tiles (4×4 in our case; it would be better to avoid this tiling, but unfortunately Inkscape isn't capable of opening much larger files, due to large physical memory consumption). In these, roads are represented as red lines and fields and forests by blue and green polygons, respectively (.svg is a vector image format).

2. Inkscape is used to convert (rasterize) the .svg files into .png files. We used the resolution of 8200×8200 texels for each of the 16 tiles.

3. Photoshop is used to batch-convert these .png files to .tga format, so the TileCombiner application can read them.

4. TileCombiner `merge` command is used to merge these tiles into a single large .tga file.

5. In our case, the large .tga file had the resolution of 32800×32800 texels, resulting in the size of 4.3GB. Photoshop won't open this file, because the .tga fileformat specification bounds the file size to 2GB at most (although the internal structure of that file is correct). Because of this, the TileCombiner `split_on_quarters` command is used to create four .tga tiles out of this single large file.

6. The two previous steps has been performed to get as large as possible tiles into Photoshop for processing. The problem is that fields and forests are generated in the form of hard-edged polygons by CityGen. To make them appear more natural, it's necessary to smooth these two layers somehow (but not the road layer!). For this we have used roughly 20-pixels-wide median filter and 3-pixels-wide Gaussian filter afterwards. Since these operations are not consistent near image edges, it's essential to have as large continuous tiles as possible for these operations, to minimize boundary artefacts during the rendering. After applying the smoothing operations, the files are saved as .tga tiles again.

7. The juggling with the tiles is reversed. First, the TileCombiner `merge_quarters` command merges the four .tga tiles back into one large file, and then the `split` command creates an array of smaller .tga tiles again (32×32 files 1025×1025 texels large in our case). These form the L1 layers.

Final combination The L0 and L1 layers are just a semiproduct, which has to be processed by the TileCombiner application to obtain the final SM0 and SM1 datasets. The processing includes calculation of the splatting map values on a per-texel basis, deciding the priority of layers and normalizing the texel, so that the sum of all 7 involved channels is always equal to 1 (otherwise colour inconsistencies on the terrain would occur). This is achieved by using the TileCombiner `combine` command. It is important that the number of tiles in both dimensions and their resolution matches. Also note that this operation requires access to the heightmaps, to determine the altitudes where fields should be clamped and forest character changes towards snowy surface. The resulting SM0 and SM1 datasets are then directly utilizable by the terrain rendering engine, or MLP, in other words. An example of these is show in Figure 27.

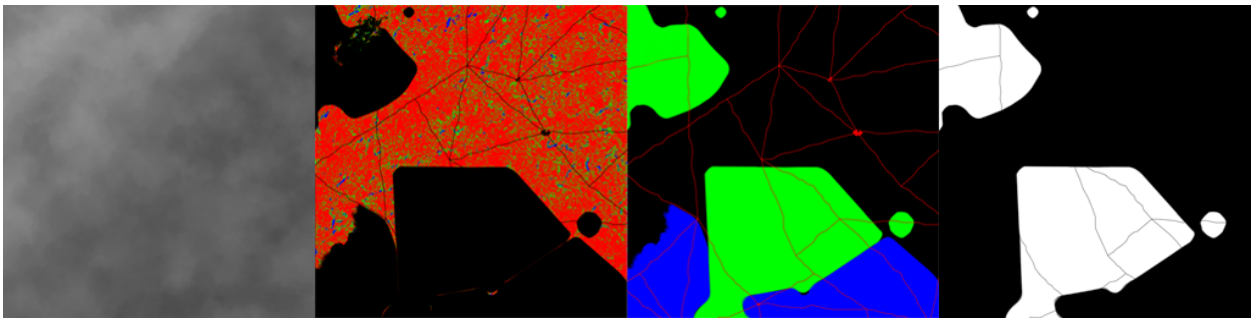


Figure 27: An example of the resulting complete dataset (except city files, which don't have a graphical representation). From left to right: heightmap, and the corresponding SM0 and SM1 splatting maps and forest density map. These correspond to roughly 14.3km by 14.3km area on the terrain.

D.3 Forest and city data

Forests and cities are commonly called 'static geometry'. They are spread over the terrain to enhance its appearance. This section does not deal with the artistic resources that represent the trees and houses themselves, just with the underlying data that are used to place these objects over the terrain.

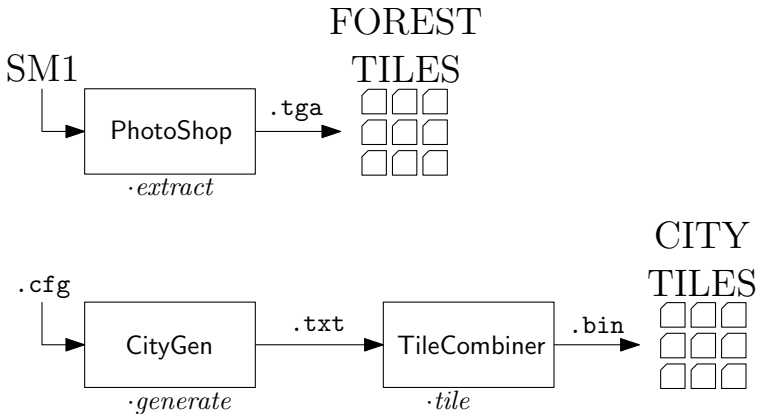


Figure 28: Static geometry data preparation pipeline.

Forest pipeline Forests, or more precisely, forest density maps, are created from the previously computed SM1's layer/channel, which corresponds to forests (green channel in our case). The procedure is quite simple – Photoshop is used again to batch process all SM1 tiles. From these, the green channel is extracted and stored separately as 8bpp greyscale .tga files with the same resolution (actually one pixel less in each dimension, that is 1024×1024 in our case; this is for practical reasons, as it's simpler to work with power-of-two files). These are then utilized by the graphical engine as described in Section 4.5.4.

City pipeline Buildings are generated by CityGen at the same time as the data for L1 layers are. CityGen produces a single .txt ASCII file containing the generated buildings data (see Section 4.5.4) depending on its configuration. This .txt file is then processed by the TileCombiner `citygen` command (the match of naming is a coincidence here), which generates an array of binary files. The buildings are divided between these

files according to their spatial positions. The .bin files are then utilized by the graphics engine during the rendering.

E Adding new models into the game

This section describes the process of adding new models into Flying Samurai. In case of airplanes, this is done in four steps, which register the data for graphical and logical subsystems of the game.

1. Create a new subdirectory of the `Media/models/` directory, named `'[airplane name]'`. Copy all necessary graphical resources into this directory, as described in Sections 4.5.2 and F.

2. Add a new resource path into the `config/resources.cfg` directory. The record should be added into the `[Airplanes]` group and should look like `FileSystem=./Media/models/[airplane name]`.

3. Add a new object record into the `Media/Objects.xml` file. It should look like this:

   ```
   <Object
     name="[airplane name]"
     GUID="[unique ID]"
     type="GOT_AIRPLANE"
     material="scripts/StarussLRM_Normal" />
   ```

   The `GUID` attribute should be an unique number across all entities in the game, not only for the airplanes in this file. Try to generate a large random number, it can be anything between 0 and 4294967295. Don't change the `type` and `material` attributes (unless you have a special material you'd like to use for the airplane).

4. Finally, add a new record into the `Entities.xml` file as a new `Airplane` node. See Section B.7 for the details.

For building models it takes one step less, because the game logic doesn't need to acknowledge them.

1. Adding the graphical resources is the same as the case of airplanes. Create a new subdirectory of the `Media/models/` directory, named `'[building name]'` and move all the necessary resources in there (one `.mesh` and two `.tga` textures, again see Sections 4.5.2 and F).

2. Again, add a new resource path into the `config/resources.cfg` directory. This record should be added into the `[Buildings]` group and will look like `FileSystem=./Media/models/[building name]`.

3. And finally, add a new object record into the `Media/Objects.xml` file. It should again look like this:

   ```
   <Object
     name="[building name]"
     GUID="[unique ID]"
     type="GOT_HOUSE"
     material="FS/House" />
   ```

   One additional condition must be fulfilled here; the `GUID` attribute must not only be unique, but it also have to agree with the `GUID` of the corresponding building in the `citytile` data (generated by the CityGen and TileCombiner tools, see Sections D.3, I and H). This unfortunately means that to add a new building, the city data and L1 splatting maps must be regenerated. A simpler way to add a new building is to substitute it for an existing one, ideally with a similar size and style. This way, the `GUID` stays the same, only a new building name and path to the resources is provided, rewriting the old one.

F Modelling and texturing conventions

The purpose of conventions for artistic content in the game is that every artist deliver their work in the same format and visual style. This is very important, and it is even more important for us, because our externists don't have an unified working environment, but rather each of them work with a different set of tools.

Meshes

Meshes, or models, represent the geometry of objects in the game. The meeting point of all modelling tools are third party exporters, which exist as plugins for all major modellers (Maya, 3DS Max, XSI, Blender etc.). These exporters export the created meshes in Ogre3D proprietary `.mesh` binary format, which is directly utilizable by the engine. All meshes used in the game are in this format.

Airplanes Most of the conventions for airplane meshes are described in Section 4.5.2, so this section will only fill the remaining holes of what haven't been defined yet.

- **Consistency** In order to work properly, all newly created airplane models must be consistent (i.e. all auxiliary geometry must be properly placed and the airplane must be divided onto correctly named submeshes). This also includes correct length measurement – one world unit (1.0) should correspond to one meter, in other word an airplane with wingspan of 10.5m should really have a wingspan of 10.5 units. This should ensure that the airplanes will have correct sizes, independently of the modelling tool they were created in.
- **Axis alignment** The left-to-right direction should be aligned with the positive X axis, the bottom-to-up direction with the positive Y axis and the front-to-back direction with the positive Z axis.
- **Primitive count** It is also important to keep the amount of geometry in moderate levels (which means about 5000–10000 triangles per airplane), since there are missions in the game where up to 20 airplanes may fly at the same time.
- **Winding order** The winding order of the geometry should be set to CCW (counter clockwise).
- **Vertex format** The following vertex format is required for airplanes (in Cg code):

```
struct AirplaneVertex {
  float3 Position   : POSITION;
  float3 Normal     : NORMAL;
  float2 UV         : TEXCOORD0;
  float3 Tangent    : TANGENT;
};
```

- **Endian** All files should be in little endian (IBM PC format).

Buildings Buildings are a part of static geometry placed on the terrain. Section 4.5.4 describes how this is done.

- **Primitive count** All buildings should have an extremely low amount of geometry, because there can be a lot of them rendered at the same time. Therefore none of them should have more than roughly 50 triangles (additional details can be of course incorporated into their normal maps).

- Otherwise, everything what have been said for airplane meshes is valid for buildings as well (consistency, axis alignment, winding order, vertex format).

Textures

Textures are image files wrapped around objects to provide 2D surface data for them, such as colour, perturbed normals, or smoothness and metalness coefficients.

Colour data RGB tristimulus values that determine surface colour (i.e. what portion of incoming light will be diffusely reflected). In our case these are three 8-bit unsigned integer values, giving 24 bits per texel (interpreted as values from 0 to 1) in R8G8B8 texture.

Surface normals Perturbing normals that add virtual details that are too fine to be modelled by the geometry. They are stored in a normal R8G8B8 texture, but the values are remapped from $\langle 0; 1\rangle$ to $\langle -1; 1\rangle$ and interpreted as three-dimensional vector.

Smoothness coefficient Special modality that determines local smoothness of an object. It is fed per-fragment into the Strauss reflectance model (which we use only for airplanes). It needs a single 8-bit channel and the values correspond to various surface roughness – value of 0 means rough (fully diffuse surface) and value of 1 means perfectly smooth surface (producing sharp highlights).

Metalness coefficient Special modality that locally determines how close is the underlying material to a metal. Values of 0 mean no metal is present on a surface, values of 1 represent fully metallic material. Along with the smoothness coefficient we use a single 16bpt L8A8 texture ('L' channel contains the smoothness values and 'A' channel contains the metalness values).

Airplane textures Airplane textures should be placed in the same directory as the corresponding `.mesh` files and `.scene` file, i.e. `Media/models/[airplane name]`. All textures have to be in the `.dds` format, either uncompressed or DXT-compressed. We use all aforementioned modalities for airplanes. The naming conventions and optimal resolutions of them are as follows:

Modality	Name	Resolution
Diffuse texture	`[airplane name]_map_diffuse.dds`	1024^2
Normal map	`[airplane name]_map_normal.dds`	1024^2
Smoothness/metalness map	`[airplane name]_map_smoothness_metalness.dds`	512^2

Building textures Building textures should be placed in the same directory as their corresponding `.mesh` file. All textures have to be in the `.tga` image format (for no special reason, just for the unification, and since they should be smaller then the airplane textures, we don't need a DXT compression for them so much). We use only colour and normal maps for buildings. The naming conventions and optimal resolutions of them are as follows:

Modality	Name	Resolution
Diffuse texture	`[building name]_diffuse.dds`	512^2
Normal map	`[building name]_normal.dds`	512^2

Splatting textures These are located in the `Media/mlp_terrain/materials/textures/splatting/` directory. They are in various formats and have various sizes, depending on the splatting material layer. Their usage is governed by the `Media/land.xml` settings file used by the Myrddin Landscape Plugin. The corresponding material settings are located in `WorldMap/Splatting/LayerMaterials` node – these determine which splatting layer uses which diffuse texture and if detail texture and normal map are applied as well. The directory contains more textures that are currently in use, so the user can experiment with them by setting different textures in the `land.xml` file, which causes the terrain to look differently. Since Ogre3D can read all common image formats, users are encouraged to add more textures here and experiment with the settings further.

G Graphical subsystem settings

Graphical subsystem is configured from the `graphics.cfg` file in terms of various constants that are used through this module. The internal structure of `graphics.cfg` is equal to XML. On start-up, this file is loaded and the values are translated into the `GraphicsEngineSettings` class, where the `GraphicsModule` class accesses them during runtime.

This section lists and explains the settings that `graphics.cfg` contains. We have tried to decouple as many constants as possible from being hardwired in the code, but in some cases this is not meaningful, so we did this only for those settings which can really become a subject to change later.

`KeyDay` (default value `0.51`)
: HDR 'Key' value during the day (subjective photometric intensity).

`KeyNight` (default value `0.04`)
: HDR 'Key' value during the night.

`WorldPageSide` (default value `14336.0`)
: Size of one terrain page in world space in meters (all pages are squared).

`WorldPageHeight` (default value `2000.0`)
: Height scale coefficient of the terrain in meters (peak terrain elevation).

`WorldTreePGPageSide` (default value `112.0`)
: Size of one forest page in meters.

`WorldBuildingsPGPageSide` (default value `14336.0`)
: Size of one page of static buildings in meters.

`CameraFreeLookConstant` (default value `0.03`)
: Controls the speed of orbiting of the free-look camera around the player's airplane.

`MaxForestDensity` (default value `9.0`)
: Peak forest density coefficient (amount of trees generated on a 14×14 meters area in full forest on highest settings).

`MinForestDistance` (default value `300.0`)
: Minimal rendering distance for forests in meters (lowest graphical settings).

`MaxForestDistance` (default value `3000.0`)
: Maximal rendering distance for forests in meters (highest graphical settings).

`MinForestBlendWidth` (default value `200.0`)
: Minimal blending width for forests in meters – transition from full density to zero density at the edge of the current rendering distance (lowest graphical settings).

`MaxForestBlendWidth` (default value `800.0`)
: Maximal blending width for forests in meters (highest graphical settings).

`FlagBillboardSize` (default value `3.0`)
: Size of the nationality-identifying flag billboard in meters.

`FlagBillboardOffset` (default value `5.0`)
: Offset of the nationality-identifying flag billboard in meters from the airplane.

`TerrainPageSide` (default value `513`)
: Discrete resolution of one terrain page.

`TreePageSide` (default value `1024`)
Discrete resolution of one forest page.

`NumPagesX` (default value `32`)
Number of pages in horizontal direction across the entire terrain.

`NumPagesZ` (default value `32`)
Number of pages in vertical direction across the entire terrain.

`ProceduralPageLimit` (default value `10`)
Size of the FIFO page cache of the `SurfacePageLoader` class.

`NumTreeTilesX` (default value `128`)
Number of forest pages in one terrain page in horizontal direction.

`NumTreeTilesZ` (default value `128`)
Number of forest pages in one terrain page in vertical direction.

`TreeTileSide` (default value `8`)
Number of tiles in one forest page (one tile corresponds to one texel in the forest page datafile).

`NumBuildingTilesX` (default value `32`)
Number of city pages in horizontal direction for the entire terrain.

`NumBuildingTilesZ` (default value `32`)
Number of city pages in vertical direction for the entire terrain.

`TreeImpostorResolution` (default value `256`)
Resolution of the impostor texture for one virtual orientation.

`PolarSkyboxTextures` (default value `5`)
Number of available skybox textures.

H TileCombiner application

TileCombiner is our own auxiliary application for manipulation with .tga images, specifically splitting them onto smaller tiles and merging these tiles back, and calculating the final splatting maps from pre-prepared semiproducts. It provides fictions in the terrain preparation pipeline (see Section D) which we couldn't find a third party software for.

The application works with uncompressed BGRA Truevision Targa .tga image files according to their 2.0 specification, without extension metadata [28]. That means we support only 4-channel .tga files with the standard 18-byte header and 26-byte footer. This is on of the standard formats produced by Photoshop or any other image editors. We don't support .tga files according to their full specification, because it serves us only as an intermediate format to manipulate with the splatting maps. We chose this format because of its simple structure and because it is widespread and well established.

Since this is our internal tool, no error checking of any kind is performed. It is assumed that all files have the previously mentioned format and all of them have sensible dimensions (i.e. splitted file must have dimensions allowing exact splitting onto the demanded amount of tiles, merged tiles must all have the same dimensions etc.), given paths are valid, no files are missing and so on. Failure to comply with these assumptions may result in undefined behaviour (that is, it will most probably crash).

The following operations are available:

split Splits one .tga file onto a 2D array of smaller .tga tiles. Syntax:

```
tile_combiner split <input image> <result dir> <# tiles X> <# tiles Y> [<bpp
    (1-4)>] [<# rows at once>]
```

- `<input image>` path to the image to split
- `<result dir>` directory where the tiles will be placed
- `<# tiles X>` number of tiles in horizontal sense
- `<# tiles Y>` number of tiles in vertical sense
- `[<bpp (1-4)>]` bit depth of the image. Optional parameter, for debugging purposes only
- `[<# rows at once>]` number of concurrent rows to process. Optional parameter, for debugging purposes only, the application determines this automatically

merge Merges a 2D array of tiles into a single .tga file. Syntax:

```
tile_combiner merge <source dir> <output image> <# tiles X> <# tiles Y> [<bpp
    (1-4)>] [<# rows at once>]
```

- `<source dir>` directory where the tiles are located
- `<output image>` path to the merged image
- `<# tiles X>` number of tiles in horizontal sense
- `<# tiles Y>` number of tiles in vertical sense

- [<bpp (1-4)>] bit depth of the image. Optional parameter, for debugging purposes only
- [<# rows at once>] number of concurrent rows to process. Optional parameter, for debugging purposes only, the application determines this automatically

split_on_quarters Splits the given file on 4 quarters with 1-texel overlap. Syntax:

```
tile_combiner split_on_quarters <input image> <result dir>
```

- <input image> path to the image to quarter
- <result dir> directory where the 4 tiles will be placed

merge_quarters Merges 4 tiles with 1-texel overlap into a single .tga file. Syntax:

```
tile_combiner merge_quarters <source dir> <output image>
```

- <source dir> directory where the 4 tiles are located
- <output image> path to the merged image

combine Combines two series of tiles corresponding to L0 and L1 layers. Creates the final SM0 and SM1 splatting maps (see Section D.2). Syntax:

```
tile_combiner combine <source dir L0> <source dir L1> <heightfield dir> <result
    dir> <# tiles X> <# tiles Y> <heightfield page side> [<tile prefix>]
    [<field clamp height (0.0-1.0)>] [<forest clamp height (0.0-1.0)>] [<# rows
    at once>]
```

- <source dir L0> directory where the L0 tiles are located
- <source dir L1> directory where the L1 tiles are located
- <heightfield dir> directory where heightfield tiles are located. There have to be the same amount of these as of L0/L1 tiles
- <result dir> directory where the resulting splatting maps SM0 and SM1 will be placed
- <# tiles X> number of tiles in horizontal sense
- <# tiles Y> number of tiles in vertical sense
- <heightfield page side> dimension of the .raw heightmap tiles
- [<tile prefix>] prefix that will be added at the beginning of the tiles' names. Optional parameter
- [<field clamp height (0.0-1.0)>] relative clamping altitude for fields. All field texels above this altitude will be nulled. Optional parameter, the default value is 0.35
- [<forest clamp height (0.0-1.0)>] relative clamping altitude for forests. All forest texels above this altitude will be converted to snow layer. Optional parameter, the default value is 0.495
- [<# rows at once>] number of concurrent rows to process. Optional parameter, for debugging purposes only, tha pplication determines this automatically

citygen Generates binary tiles of buildings from the single .txt file generated by CityGen. Syntax:

```
tile_combiner citygen <input file> <result dir> <# tiles X> <# tiles Y> <tile
    world size>
```

- `<input file>` path to the .txt file that contains building data
- `<result dir>` directory where the .bin tiles will be placed
- `<# tiles X>` number of tiles in horizontal sense
- `<# tiles Y>` number of tiles in vertical sense
- `<tile world size>` size of one tile in world units (we use 14336.0)

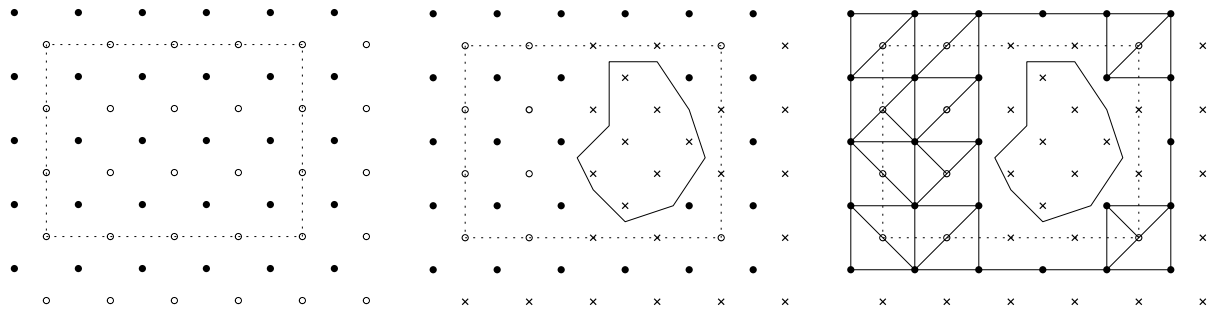
I CityGen application

To make the terrain more realistic, fields, forests, cities and roads have to be generated (see Section 4.5.3). It was originally our intention to use InkScape[14] to draw all the roads and generate the cities based on their positions and size. This has, however, proved to be a very unrealistic idea, as the availability of historical maps is rather bad and the number of roads to draw is too large. Therefore, we have decided to generate both the roads and the cities randomly, adding fields and forests for further realism.

The CityGen tool is an ad-hoc piece of software that we programmed to make resource preparation easier and much less time-consuming. It is, by no means, a full blown procedural generation framework; due to it's complexity and it's one-off nature, making this tool robust was hardly our priority.

I.1 Generation of road network

First, a regular grid of vertices – future villages, town, and cities – is generated (see `!Grid::Initialize`). Then a copy of this grid, translated by half the distance between (horizontally or vertically) neighbouring vertices is generated; those vertices are called dual vertices (see image 29(a)). Those are then checked against a list of forbidden, so called Chernobyl, polygons where there are no cities or roads allowed. Those in the Chernobyl zones get discarded (see image 29(b)). Also, each dual vertex checks if the four neighbouring non-dual vertices aren't discarded. If even just one of those has been discarded, the dual vertex gets discarded too. This greatly simplifies the possible cases of the next step.



(a) Vertices and dual vertices (b) Chernobyl and discarded vertices (c) Road network

Figure 29: Map border is in a dotted line, (non-dual) vertices are filled, dual vertices are just outlined, Chernobyl polygon(s) is in a full-line, and vertices, discarded either due to not having enough neighbours or due to being in the Chernobyl polygon(s) are marked as crosses. Notice that no dual vertex chose to discard itself in Figure 29(c)

In the next step, each non-dual vertex spawns a road to it's left and to it's right (if the potential neighbour still exists). Each dual vertex randomly chooses one of the following four cases:

Discard Discards the city and spawn no roads whatsoever.

Diagonal Spawns two roads that connect the dual vertex' neighbours in a diagonal fashion (one of the two possible diagonal is picked).

Three out of four Builds roads to all but one of the four non-dual neighbours.

Four out of four Builds roads to all of the (four) non-dual neighbours.

As a result (see image 29(c)), a plausible, although not absolutely prefect, road network is generated. Some artefacts still remain (see the bottom and top left corners of image 29(b)).

Each vertex position gets perturbed and each road connecting two vertices gets subdivided and it's vertices sub-positions get perturbed, too (see `PostprocessRoads`). This results in a much more believable road network. For clarity sake, this step has been omitted from Figure **??**.

I.2 Generation of cities

For each of the vertices (both dual and non-dual), it's degree is calculated (see `GenerateCities`). If the degree is higher than a constant (currently 4) or exactly one (to avoid blind roads), a random city area a is generated using the following formula:

$$a = r^p \cdot (\text{side}_{max} - \text{side}_{min}) + \text{side}_{min}$$

where $r \in \langle 0, 1 \rangle$ is a random number, side_{max} and side_{max} are the minimum and maximum city sizes, respectively, and p is the `citySideRemapPower` parameter (see below) that introduced non-linearity to city area. From the area, the city size is chosen randomly (see `CityRect::CityRect`)

After that, intersections of all roads with the city's boundary are made and all edges that intersect the city's bounding box get stored in a minimap (see Figure 30).

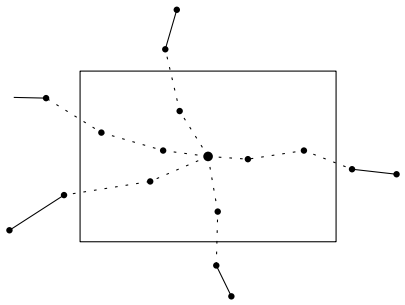


Figure 30: Minimap; the rectangle denotes the city's bounding box, dashed lines are part of the city's minimap, full lines are not, small points are road subdivision vertices, big vertex is the original vertex from the road network generation phase.

This minimap is then used as a basis for city generation (see `GenerateCity`). All of it's roads are recursively subdivided until a limit length for each segment is reached. All of the vertices within the minimap are now enqueued as spawn points (see `spawnPoints` in `GenerateCity`). From each spawn point (a spawn point is a position and the direction in which a new road will be generated), a new road that is roughly perpendicular to the original road can be generated.

If this new road ends outside the ellipse that fills the bounding box, it gets discarded (see Figure 31).

If an intersection with another road is found, the road's two intersections are checked and if one of them is within a given limit, the newly spawned road's endpoint gets snapped to it (if it doesn't intersect other roads, if it does, it gets discarded). If neither of the road's two intersections is close enough, the intersection is used as the newly spawned road's endpoints. The spawn point gets deleted from the spawn point queue.

If no intersection with another road is found, the nearest road endpoint (belonging to a different road, obviously) is found; if the road from the original spawn point to the nearest vertex doesn't intersect any

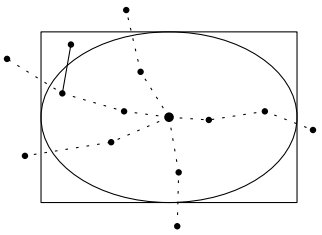


Figure 31: The ellipse filling the minimap; it is used to make the cities rounder; the newly spawned road, depicted by a solid line, would get discarded.

other roads, it gets added. In any case, the spawn point gets discarded.

Finally, if there was no intersection and no other endpoint was close enough to snap to, a new "blind" road gets generated. The (new) endpoint of this road spawns 0 to 3 new points, each with a different spawn direction. The probability of a new spawn point with a respective spawn direction (left, right, forward with respect to the existing road) influences the shape and density of the road network structure.

When there are no spawn points left, the road network generation ends.

I.3 Generation of buildings

All of the resulting street are then iterated (in no particular order), and each of the streets gets populated (see `PopulateStreet`). From one end of the street to the other, new random buildings (see Section I.6 for information about how buildings are defined) are generated on each side of the road and an offset that denotes the already occupied street length is kept for each side. Before any buildings gets added, it gets checked for intersections with other buildings and roads. If there is an intersection, a constant value is added to the occupied street length to make sure that the algorithm stops and to add the ability to skip occupied places (see Figure 32).

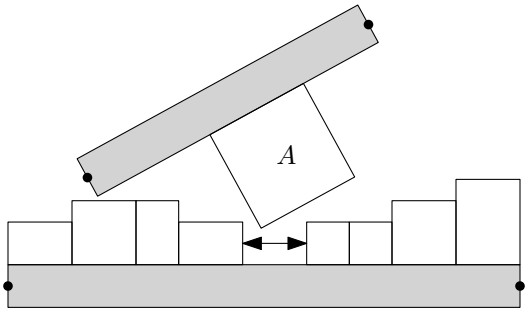


Figure 32: Roads in grey; the buildings are added from left to right (by convention). Each time a building is added, the occupied street length gets increased by the amount taken by the newly generated building. Notice the free span that has been skipped; this was caused by (already existing) building A occupying space in the vicinity of the road. If a building that could fit (without collisions) into the free space doesn't get generated, a certain amount of space gets skipped (repeatedly if needed).

There are three classes of building types: village buildings, town buildings, and city buildings. Depending on the area of the generated city, the city is marked either as a village (village buildings only), town (town

buildings and village buildings), or a city (all three kinds of buildings). The city type then acts as the limit for the best building type. Building types form a gradient with respect to distance from the city centre (see Figure 33. For each building type, several building subtypes might exists. After the building type is determined, a subtype is randomly chosen.

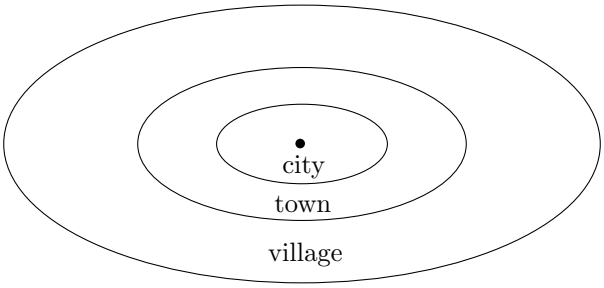


Figure 33: Buildings gradient for a city of city type.

I.4 Generation of fields, forests, and occluders

To add more detail to the terrain, fields and forest are also generated. For both fields and forests, a regular n-gon is generated and each of it's vertices is then displaced along the line connecting the vertex to the centre. Forest polygons get discarded if there's a city within their bounding box.

To add irregularity to both fields and forests, occluder polygons were introduced. Also regular n-gons with displaced vertices, occluders get drawn in the colour of grass and thereby create dents and holes in both cities and fields. To create some free space around cities, an occluder is also added under each city.

I.5 Output

Due to InkScapes instability, output has to be split up into smaller chunks (see the `Cell` class). For each of the chunks, a list of all items that overlap it is generated and each chunk is then output separately.

Output colours are currently hard-coded into the tool's source. Fields are pure blue, forests are pure green, grass and occluders are black, and roads are red. The output can be seen in Figure 27, third picture from the left.

Due to the rather primitive implementation, the tool is rather time demanding. Be ready to wait a few hours for larger terrains.

I.6 Configuration file

The whole tool is configured using the `config.xml` file. In the root tag (`<config>`), the output file and the Chernobyl SVG files are specified

```
<output filename="cityout2.svg" debug="true"/>
<chernobyl filename="forbidden2_polys.svg"/>
```

If the `debug` attribute is set to `true`, auxiliary output files with debugging information will be output.

The Chernobyl file is an InkScape SVG file that contains polygons. Those are saved as polylines by InkScape. All polylines whose first and last point are equal will be used as forbidden polygons (see `LoadPolygons`). The file's content will automatically get scaled to the dimensions specified in the `config.xml` file.

Then, a `parameter` section follows.

```
<parameters>
    <parameter name="width" value="458752"/>
    ...
</parameters>
```

Below is a list of possible parameter names[2]. All dimensions are in meters or in meters squared. See the default configuration file for an example.

width width of the map to be generated (overrides SVG dimensions!)

height height of the map to be generated (overrides SVG dimensions!)

scaleFactor by what factor to scale contents of the file.

subdivisionThreshold divide streets under this length, in meters (recommended value: 50).

subdivisionVariation when dividing a street, how regular the division should be (+- 0.2 of the length of the original street).

streetWidth how wide should the streets be (needed for building positioning, doesn't affect SVG)

roadWidthSVG width of the roads/streets for SVG export.

streetDeviation how much should each coordinate of a new street be perturbed; ± value meters.

vertexSinkSize when loading SVG data, merge vertices closer than value meters.

spawnLengthMin minimum length of generated streets (servers as a hint, might be shorter).

spawnLengthMax maximum length of generated streets (servers as a hint, might be longer).

townArea classify cities with an area ≥ of value m^2 as towns.

cityArea classify cities with an area ≥ of value m^2 as cities.

forrestRadiusMin] minimum forest radius hint forrestRadiusMax] maximum forest radius hint forrestRadiusRemapPower] remapping parameter of the forest radius, similar to "gamma" parameter in gamut mapping equation; the actual forest radius r is determined by the equation

$$r = rMin + (rMax - rMin) \cdot unitRand^{r}emapPower$$

and so value of `remapPower` of 1.0 gives no bias, values < 1.0 produce larger forests (likely closer to `rMax`) and values > 1.0 produce smaller forests (likely closer to `rMin`); The *unitRand* variable is a random number $\in \langle 0, 1 \rangle$.

forrestRadiusVariation between 0.2 and 0.8, influences the forest shape (smaller -¿ more round, higher -¿ more rugged).

forrestCount how many forests should be generated (note: many forest will get merged, some will get discarded; this value is just a hint) .

[2]Forest is sometimes incorrectly spelled as forest; use the incorrect version in this file

fieldRadiusMin see `forrestRadiusMin`.

fieldRadiusMax see `forrestRadiusMax`.

fieldRadiusRemapPower see `forrestRadiusRemapPower`.

fieldRadiusVariation see `forrestRadiusVariation`.

fieldCount see `forrestCount`.

occluderRadiusMin see `occluderRadiusMin`.

occluderRadiusMax see `occluderRadiusMax`.

occluderRadiusRemapPower see `occluderRadiusRemapPower`.

occluderRadiusVariation see `occluderRadiusVariation`.

occluderCount see `occluderCount`.

horizontalTiles in how many cells/chunks/tiles should the terrain be split horizontally?

verticalTiles in how many cells/chunks/tiles should the terrain be split vertically?

citySideMin minimum city side (60 - small, 150 - moderate, 500 - maximal reasonable thing (count with approximately 20 seconds per town!).

citySideMax maximum city side.

citySideRemapPower see `forrestRadiusRemapPower`.

cityOccluderCoef scale factor for the city occluder, even values under 1.0 *might* be safe (there's some margin).

Finally, after all parameters, a database of available building types follows. In the `<Buildings>` tag, various building subtypes are defined.

```
<buildings>
    <building width="10" height="10" GUID="435" type="village"/>
    ...
    <building width="15" height="15" GUID="637" type="town"/>
    ...
    <building width="20" height="20" GUID="535" type="city"/>
</buildings>
```

The width is the size of the building/parcel from east to west, the height is the size of the building from north to south, orientation is with respect to south, GUID (see Section 4.5.8).

J PathGen

PathGen is a simple tool written in C# that converts InkScape's poly-lines into the game's XML format and generates airfield positions. It takes an InkScape SVG file (not a regular SVG!) as input. See the `template.svg` for an example of

Bring up the "Layers" → "Layers..." panel in InkScape. Add new layers and name them $front, YYYY - MM - DD$, where $YYYY - MM - DD$ is the date from which the front is valid (see Figure 34 for an example). In each of those layers, draw a poly-line representing the layer as it was at the specified date.

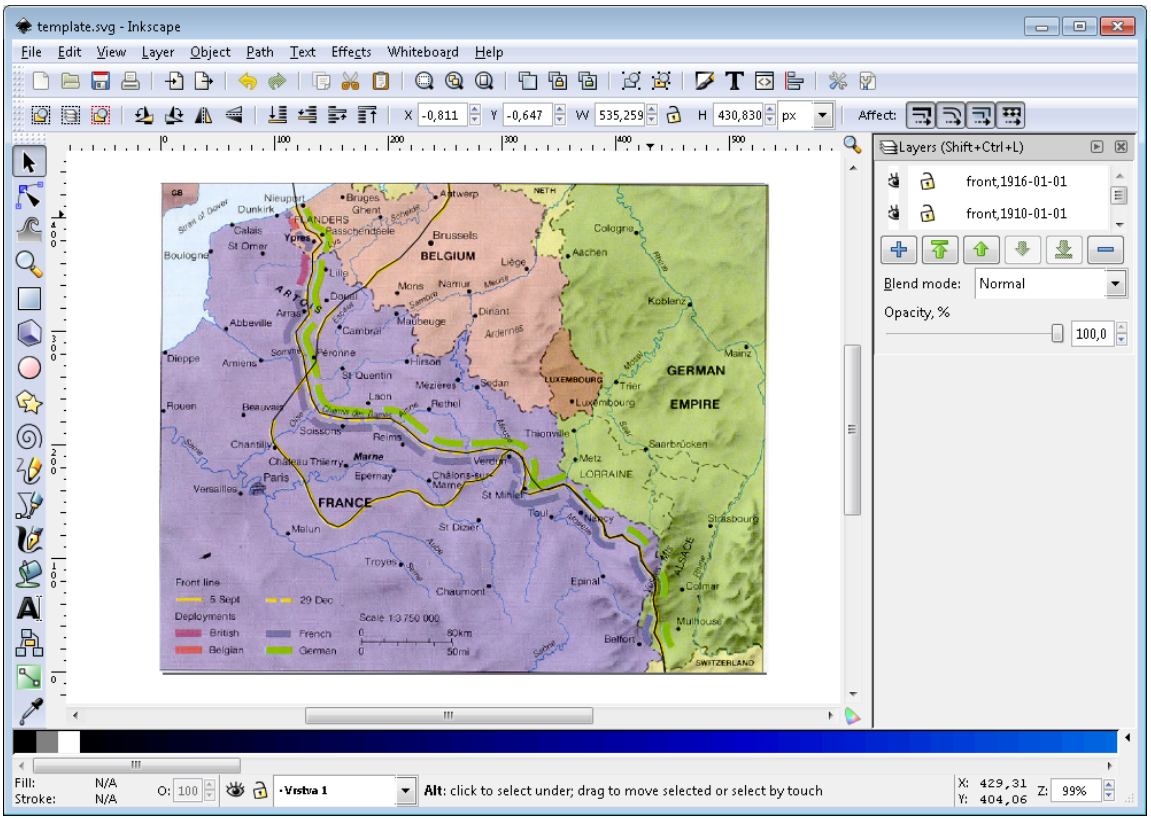


Figure 34: The "Layers..." panel and the `template.svg` file in InkScape. Notice the front layers definitions on the right side. On the left side, two black-colour lines represent the two fronts.

Such an SVG file is then selected in the PathGen application ("Inkscape's XML file"). Several parameters can be specified in the dialogue. If you also want airports to be generated, specify the minimal and maximal distance and airplane should be from the front, the number of airfields (for each side), and the nation IDs (as defined in Section B). After pressing the Convert button, extract the relevant part of the XML that is output into the textbox at the bottom of the dialogue.

This is a tool. It will give you a set of airfields, but they are not guaranteed to be on the correct sides of the front (if the front has a lot of turns, the normal from the previous front segment might intersect the next segment, thus putting the airfield on enemy territory) , although, in most cases, they will be. Use it to pregenerate data that you then review manually.

The implementation of this tool is rather straightforward. The input file is parsed, and the fronts are

converted. Then, a set of airfields is generated with positions that change with the front and with time. The export to XML is done using .NET's in-built framework.